Flexible Concurrency Control by Reasoning about Database Queries and Updates

Charles P. Elkan

Ph.D Thesis

TR 90-1145
August 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

FLEXIBLE CONCURRENCY CONTROL BY REASONING ABOUT DATABASE QUERIES AND UPDATES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Charles Peter Elkan

August 1990

FLEXIBLE CONCURRENCY CONTROL BY REASONING ABOUT DATABASE QUERIES AND UPDATES

Charles Peter Elkan, Ph.D.

Cornell University 1990

A number of database management problems involve reasoning about queries and updates. Concurrency control is the most important example: two transactions should not be executed simultaneously if it is possible that an update command issued by one transaction might change information used in answering a query issued by the other. Existing concurrency control schemes are based on the idea of protecting discrete items of data. This thesis describes a concurrency control scheme called adaptive locking that is based instead on logical reasoning.

The central notion is that of *independence*. Informally, a query is independent of an update if executing the update cannot change the result of evaluating the query. First the general properties of the concept of independence are investigated, using a formal model-theoretic definition in the context of deductive databases. Then proof-theoretic sufficient conditions are obtained for the independence of queries and updates. These results apply to arbitrary queries and updates, and they take into account integrity constraints and recursive rules.

For the special case where a query and an update are both specified by conjunctive relational algebra expressions, a decision procedure for independence is given. The procedure is of practical use because typically it requires linear time, and it produces answers that are precise enough to be relied upon. The procedure takes into account functional dependencies, so it constitutes a solution to an open problem identified by Blakeley, Coburn, and Larson. It is of theoretical interest for two reasons. First, its quadratic worst-case time complexity cannot be improved unless the reachability problem for directed graphs can be solved in sublinear time. Second, it applies to the widest possible natural class of queries, since deciding independence is $\mathcal{NP}$-hard for nonconjunctive queries and updates.

Biographical Sketch

Charles Elkan was born on March 17, 1963 in Wellington, New Zealand. In 1970 he moved with his family to Geneva, Switzerland, where in 1981 he graduated from the Collège Rousseau with two *maturité* diplomas. He then went up to Fitzwilliam College at Cambridge University, going down in 1984 with a B.A.(Honours) degree in mathematics. From 1984 to 1989 he was a graduate student in the Department of Computer Science at Cornell University, except for six months in 1986 at Stanford University, three months in 1986 at ATT Bell Laboratories, and four months in 1988 at Xerox Palo Alto Research Center. Charles Elkan spent the 1989/90 academic year at the University of Toronto as a (pre)postdoctoral fellow. In September 1990 he will become an assistant professor in the Department of Computer Science and Engineering at the University of California, San Diego.

Acknowledgements

I want to thank my parents for making me a citizen of New Zealand and then taking me to Switzerland. Any true education that I possess comes from my family and the Collège Rousseau.

The faculty of the Department of Computer Science at Cornell have created an environment that is wonderfully scholarly, friendly, and enterprising. My committee members, official and unofficial, have been inspiring examples—Prakash Panangaden, Dexter Kozen, David McAllester, and Alberto Segre. Thank you for your confidence and for your patience.

My friends in Ithaca and in California helped me see more of life each year. From my first years at Cornell I want especially to thank Michael Schwartzbach, Patrick Stephenson, Wilfred Chen, Yang Ro Yoon, and Brad Vander Zanden. For help in the final weeks of writing this thesis, I want to thank Kieran Herley, Kjartan Stefansson, and Nils Klarlund.

And then I must thank George Staller and Kasia Andrzejewska for helping me stay sane in the new world, for reminding me of the old world, and for much more.

Table of Contents

List of Illustrations

Chapter 1

Introduction

The design of database systems is one field of computer science where academic research has had a significant impact on industrial practice. The two principal research achievements have been the invention and refinement of relational database systems, and the development of protocols enabling multiple users to access a shared database concurrently [NSF, 1990]. Relational database theory has yielded query languages that combine efficient implementability and a clear semantics with the flexibility to access data in ways that need not be anticipated when a database is established. The theory of transaction management provides methods for guaranteeing that multiple users of a database system do not interfere with each other.

Remarkably, although the theory of relational databases supplies languages based on first-order logic for specifying database access operations and the core issue in concurrency control is reasoning about interference between access commands issued by different users, no mainstream work on concurrency control uses logical reasoning about queries and updates. That is the topic of this thesis. The aim is to link the two major achievements of past research on database systems and to lay a foundation for the development of more sophisticated database systems.

The first section of this introductory chapter provides an extended example of the type of logical reasoning about queries and updates advocated here. The second section summarizes the contributions of this thesis, and outlines its contents chapter by chapter.

1.1 A motivating example

Typically a number of updates to a database are required to reflect a single conceptual change in the external world. Such a collection of related updates is called a transaction. Users want transactions to be executed in their entirety: if some component of the database system crashes during the execution of a transaction, then those updates in the transaction that have already been executed should be undone, in order to restore the database to a consistent state. Moreover, even if a transaction is not cancelled, while it is being executed other users should not be able to retrieve information that it has updated, since this information may be subject to further updates. There are yet more consistency concerns to take into account; in general, update and query commands issued by different users cannot always be executed in the same order that they are issued. To ensure that each user sees the shared database in a consistent state, access commands must often be delayed, or even rejected.

The reasoning necessary to decide when it is safe to allow different users simultaneous access to part of a database can be complicated. For example, imagine a relational database system storing information about stockbrokers, their offices, and their customers. (This database might be maintained by a regulatory agency.) Suppose that the database contains, among others, two relations respectively connecting brokers with the names of their customers, and with the locations of their offices. These relations can be specified using a tabular data definition language as follows.

name	*schema*	*interpretation*
client	PB	person P is a client of broker B
office	BC	broker B is based in city C

Suppose further that every person is known to be a customer of at most one broker, and that every broker has an office in at most one city. (These assertions are not realistic in the United States in the 1990s, but they might well be legally enforced in some other jurisdiction.) Constraints of this type are called functional dependencies. Now suppose one user of the database system wants to update the information recorded in the client relation by deleting all $\langle P, B\rangle$ tuples such that the office of the broker B is located in Los Angeles. Using relational algebra notation, this update can be expressed as

$$\textsf{delete client:}\ \pi_{PB}\left(\sigma_{C=Los\ Angeles}(\textsf{client}\bowtie\textsf{office})\right).$$

Suppose that at the same time another user wants to see a list of all customers of brokers whose offices are in New York. This query can be expressed as

$$\textsf{read}\ \sigma_{C=New\ York}(\textsf{client}\bowtie\textsf{office}).$$

Is it safe to answer this query immediately? Or must the query be delayed until it is certain that the update will not be rolled back?

The answer is that the query may be executed immediately, because undoing the update would not change the answer to the query. Informally, this can be argued as follows.

> "Suppose $\langle P_1, B_1, C_1\rangle$ is a tuple in the answer to the query. Then $C_1 = New\,York$. Now suppose $\langle P_2, B_2\rangle$ is any tuple deleted from the relation client. Because of the join operation that was performed to find which tuples to delete, there must exist a matching tuple $\langle B_2, C_2\rangle$ in the relation office with $C_2 = Los\,Angeles$. Clearly $C_1 \neq C_2$ always. Every broker has a unique office location, so $B_1 \neq B_2$. Every person

has a unique broker, so $P_1 \neq P_2$. Therefore, if $\langle P_1, B_1, C_1\rangle$ is in the answer to the query, it is present whether or not $\langle P_2, B_2\rangle$ is removed from client."

Reasoning about independence can serve a second purpose. Suppose the database system is distributed, with sites capable of executing access commands located in California and in New York. Let the update command

$$\text{delete client: } \pi_{PB}\left(\sigma_{C=Los\ Angeles}(\text{client} \bowtie \text{office})\right)$$

be issued by a user located in California. If the New York site does not store copies of the whole of the client and office relations, but only the view defined by the expression

$$\sigma_{C=New\ York}(\text{client} \bowtie \text{office}),$$

then it need not be informed of the California update. Logical reasoning about queries and updates can sharply reduce the amount of information that must be transmitted between the different sites of a distributed database system.

1.2 Outline

The main contribution of this thesis consists of practical methods for deciding when a query is independent of an update. Inference rules are given for the general case, and an efficient algorithm is given for the special case of queries and updates defined by conjunctive relational algebra expressions. The algorithm typically uses time linear in the length of the text of a pair of expressions. It is thus efficient enough to be applied online in a concurrency control scheme called adaptive locking where before executing each access command, the database system checks by logical inference whether the command causes a conflict.

This thesis has been purposely kept brief, in the hope that its readership will thereby be wider. Although the thesis is intended to be a self-contained and comprehensible document, it does not include detailed expositions of the previous work by others that was the starting point of the research presented here. Instead, I have preferred to cite sources that I consider well-written and understandable. A reader who wants to get a quick impression of the results of this work can skim through the chapters, looking only at what I have called "illustrations." These are boxes placed throughout the text that contain examples, and sometimes also explanatory diagrams. Each illustration is accompanied by notes that are intended to make it relatively independent of the surrounding text.

The thesis is organized as follows. Chapter 2 provides a perspective on the design of schemes for controlling the concurrent execution of transactions. It makes a distinction between "item-oriented" and "value-oriented" concurrency control, arguing that the latter have important potential benefits. Chapter 2 also discusses difficulties in implementing value-oriented schemes that are overcome in this thesis.

Chapter 3 investigates the concept of independence at a high level of abstraction. Deductive databases are presented formally, independence is defined precisely, and basic properties of the concept are proved. Special inference rules are justified for reasoning about independence of updates and conjunctive and recursive queries.

Chapter 4 presents a procedure that can be used to decide independence for queries and updates specified by conjunctive relational algebra expressions. The decision procedure works by applying three conditional rewrite rules to conjunctive expressions represented using a new tableau notation. Chapter 5 proposes a good representation for tableaux and a corresponding algorithm for doing all applications of the rewrite rules efficiently.

Chapter 6 shows how to use the results of previous chapters in a new concurrency control scheme called "adaptive locking." The scheme is compared qualitatively to traditional locking schemes, and a transcript shows it working in a prototype implementation. Finally Chapter 7 suggests how independence reasoning could be applied in a fully distributed database system, and discusses the significance of the results of the thesis.

Chapter 2

Issues in concurrency control

The database systems to which the work reported in this thesis applies include both traditional centralized relational database systems, and also more complex distributed and deductive database systems. In a distributed database system executing a single query or update may require the cooperation of processing sites that are dispersed over a wide area; in a deductive database system logical inference following rules may be required in addition to data retrieval. Whatever the type of database system, when multiple users may call for queries and updates to be executed concurrently, it is necessary to enforce a protocol that limits concurrency, in order to preserve consistency. The main contribution of this chapter is a new perspective on the issues involved in designing concurrency control protocols.

Section 2.1 summarizes the traditional perspective. Section 2.2 then discusses the distinction between item-oriented and value-oriented concurrency control, and Section 2.3 argues that value-oriented schemes are potentially superior on a number of criteria. Section 2.4 describes the original work on value-oriented concurrency control, and the difficulties that caused the idea to be abandoned. Section 2.5 describes some partially successful previous proposals to overcome these difficulties. The remainder of the thesis is devoted to fully overcoming the difficulties.

2.1 The traditional perspective

The scheduler of a database system is its component that manages the sequence in which query and update commands issued by multiple users are executed. This section presents the traditional view of the functionality required of the scheduler of a database system. The terminology introduced here is standard, and similar to that used in [Ullman, 1982; Maier, 1983; Bernstein *et al.*, 1987]. As far as possible, this chapter stays at a level of abstraction where the language used to express queries and updates is irrelevant. The subsequent chapters specifically use relational algebra or first-order logic as query languages.

2.1.1 Access commands

The main service that a database system provides to its users is to execute access commands that they issue. These are of three types. A *query command* has the form

$$\mathsf{read}\ \alpha$$

where α is a query expression. Whatever the language in which it is written, a *query expression* is an intensional description of some data. When a database system executes a query command, it evaluates the query expression and presents the data described by the query expression in an explicit, extensional form.

The *state* of a database is the exact information stored in it at a certain time. The current state is the cumulative result of all past updates. The state of a relational database consists of a set of tuples for each relation in the database schema. The value of a query expression is, in general, different for different database states.

Update commands may change the current state of the database by adding information or by removing information. An *insert command* is written

$$\mathsf{insert}\ \mathsf{R} : \alpha$$

while a *delete command* is written

$$\mathsf{delete}\ \mathsf{R} : \alpha.$$

In each case R is the name of a relation, and α is a query expression. The command $\mathsf{delete}\ \mathsf{R} : \alpha$ is executed by removing from the current extension of the relation named R the set of tuples that is the result of evaluating the expression α. Note that it is not necessary to require that every tuple removed actually appear in R beforehand. Conversely, the command $\mathsf{insert}\ \mathsf{R} : \alpha$ is executed by adding the result of evaluating α to the extension of R.

Access commands are defined above from the point of view of a database system. From the point of view of a user, the commands must be extended to allow information to be exchanged with local program variables. Insert and delete commands can then be combined to achieve more complex update operations. For example, Carl Sagan's salary can be doubled by the sequence of commands

$$\begin{aligned}
\mathsf{read}\ \sigma_{P=\text{“Carl Sagan”}}(\mathsf{salary})\ \mathsf{into}\ \bar{t} \\
\mathsf{delete}\ \mathsf{salary} : \sigma_{P=\text{“Carl Sagan”}}(\mathsf{salary}) \\
\bar{t}[S] \ \leftarrow\ 2 \cdot \bar{t}[S] \\
\mathsf{insert}\ \mathsf{salary} : \bar{t}
\end{aligned}$$

The modifier $\mathsf{into}\ \bar{t}$ specifies the local variable $\bar{t}$ as the destination for the result of the query command that finds Carl Sagan's current salary. The command $\bar{t}[S] \ \leftarrow\ 2 \cdot \bar{t}[S]$ is a private operation performed by the user of the database system. The system, we assume, is not informed of these private operations. Therefore, as far as it is concerned, $\bar{t}$ denotes a constant tuple of unknown value in the final insert command.

2.1.2 Transactions and locking

Modifying one field of a tuple is an example of a single "high-level" operation on a database that must be implemented by a sequence of access commands (and some private operations). "High-level" means that the operation corresponds to a conceptually indivisible operation in the real world that the data in the database system is supposed to describe. To reflect this indivisibility, the execution of a high-level operation should be atomic: either all the component commands should be executed, or, if some component of the system fails, none should be executed. Moreover, these commands should only be executed in the sequence specified. The concept of transaction makes precise the idea of high-level operation.

From the point of view of a user, a transaction is one execution of a program containing various access commands. Formally, a transaction is a sequence of access and synchronization commands. The other activities of a transaction are private, and unavailable to the database system. The commands of a transaction are labeled with its name, which is an arbitrary unique identifier. There are three types of synchronization command:

$$T_i : \mathsf{start}$$
$$T_i : \mathsf{commit}$$
$$T_i : \mathsf{abort}$$

The first command of every transaction, introducing its name, is always start. The last command is always either commit or abort, indicating that the effects of the transaction should be made permanent, or that the transaction should be treated as never having existed. Normally abort commands are not issued by a user, but rather by the database system itself, when it becomes impossible to assure that a transaction will be able to commit.

The multiple users of a database system can initiate transactions whenever they wish. The database system must control the execution of the transactions so that they are *recoverable* and *serializable* [Bernstein *et al.*, 1987]. Recoverability is the condition that for all transactions T_i and T_j, if T_j reads a value written by T_i, then T_j does not commit until after T_i commits. The intuition here is that if T_j committed and T_i later aborted, then T_j would be incorrect.

Serializability is a more complex condition: the actual sequence of execution of access commands should be such that the effect of each command is the same as its effect in some notional execution sequence where all transactions are executed serially, with no interleaving of the access commands of different transactions. The effect of a query command in two execution sequences is the same if the result of evaluating its query expression is the same. The effect of an update command is the same if it adds or removes the same information from the database.

The rationale behind the serializability condition is that users could actually have submitted the transactions to the system according to the serial order. In that case no concurrency control would have been necessary. Executing transactions serially is correct, so a serializable execution is also correct. Explanations and proofs of these claims can be found in [Eswaran *et al.*, 1976; Bernstein *et al.*, 1987].

Formally, the input to a scheduler is an arbitrarily interleaved sequence of synchronization and access commands concerning multiple simultaneous transactions. The output of the scheduler is the access commands reordered so that their execution sequence is recoverable and serializable. This formal view of the functionality of a scheduler depends on the assumption that the execution of individual query and update commands is atomic.

The most common concurrency control protocols enforce serializability by "locking" portions of the shared database. Basic locking involves locks of two modes, called "exclusive" and "shared." A transaction requests a lock by issuing the command

$$\textsf{lock exclusive } \alpha$$

or the command

$$\textsf{lock shared } \beta.$$

In these commands α and β are expressions or names designating portions of the common database.

A scheduler following a locking protocol ensures that access commands are interleaved appropriately by delaying a transaction that requests a lock if that lock cannot be granted. A lock can be granted on a part of the common database unless it conflicts with a lock already possessed by another active transaction. Two locks conflict if the mode of at least one is exclusive, and they concern overlapping parts of the database.

A transaction is *two-phased* if all its locks are claimed before any are relinquished. A transaction is *well-formed* if each access command only retrieves or updates parts of the database that are covered by locks that it possesses. Retrieved parts of the database must be protected by shared mode locks and updated parts of the database must be protected by exclusive mode locks. It is proved in [Eswaran *et al.*, 1976] that if all transactions are well-formed and two-phased, then the execution of transactions is serializable, and hence correct.

2.2 Item- versus value-oriented schemes

Whether the scheduler of a database system uses locking or some other type of concurrency control, its principal task is to keep track of what parts of the shared database are accessed by each transaction. Almost all discussions of concurrency control make the basic assumption that each query or update command refers to an arbitrary set of mutually disjoint parts of a database, and that these have no relevant internal structure: they are discrete "items." Usually items are defined in an implementation-dependent way. For a database stored in main memory, items may be words or pages, while for a database stored on disk, they may be cylinders, tracks, or sectors. Items can also be defined in a more hardware-independent way. For a navigational database [Ullman, 1982], items can be sets or records, or for a relational database, they can be relations or tuples.

The common feature of item-oriented concurrency control schemes is that the underlying database is taken to be partitioned into separate extents of data that do not overlap. The way in which the database is partitioned into items has no connection with the way in which access commands specify data to retrieve or update. Therefore it is impossible to exploit whatever characteristics serve to establish the identity of items for concurrency control purposes. Item-oriented concurrency control schemes necessarily take the view that each access command calls for retrieving or updating in an arbitrary fashion the values of an arbitrary subset of items.

In other words, item-oriented concurrency control schemes manipulate tokens as names for sections of a database without attaching any semantics to the tokens. An item-oriented scheduler cannot perform any inferences about the logical relationships between two items, beyond deciding whether they are equal or different.

The value-oriented alternative is to designate parts of a database for concurrency control purposes using expressions in some language. In contrast to a name, which is arbitrarily attached to whatever it denotes, an expression describes what it denotes by stating constraints. Everything that satisfies the constraints, and nothing else, is denoted by the expression. For example, the expression

$$\langle X, Y \rangle : Y = \textit{Los Angeles}$$

describes the set of all pairs whose second component is "*Los Angeles*." Depending on what language is used for expressions, many relationships can be deduced between parts of a database when the parts are defined by logical expressions. For example, it is possible to infer that one expression always describes a subset of the set of tuples described by another.

Value-oriented concurrency control was originally motivated by the phantom tuple problem. This problem is that it is not possible to prevent a new tuple from being inserted into a database, if locks can only protect concrete items consisting of extents of data that are already present in the database.

2.3 Advantages of value-oriented concurrency control

The key to effective database system design and implementation is to provide alternatives. For every function that a system must implement, a number of algorithms and architectures are available. To increase the range of options in the design of other database system components, a scheduler should use a concurrency control protocol that is as insulated as possible from the details of other system components. From this point of view, compared to item-oriented concurrency control there should be a number of advantages to detecting conflicts between transactions by logical reasoning.

- First, logical reasoning about query, update, and locking commands can be performed using only the text of the commands. The expressions in the commands can be analyzed without evaluating them, so value-oriented concurrency control does not require retrieval of information from the actual database.

  In typical database systems today, the limiting constraint on overall throughput, that is on the number of transactions executed per second, is latency in accessing data randomly on disks, or low bandwidth in transferring data to and from disks. It is therefore vital that all disk accesses contribute directly to executing access commands.

- Second, logical reasoning can sometimes provide more precise determinations of potential conflict. An item-oriented scheduler cannot see exactly what information is retrieved or updated by an access command, but only what items are touched by the command. When a query and and update concern the same item, an item-oriented scheduler must delay one of the commands. If individual items are large, then such a scheduler may often prevent transactions from executing concurrently that do not in fact conflict.

By doing exact reasoning about access commands, a value-oriented scheduler can avoid delaying access commands unnecessarily. While maximizing the number of transactions allowed to proceed does not necessarily give maximum throughput, it does increase the opportunities for another component of a database system to manage the degree of multiprogramming.

- Third, with value-oriented concurrency control conflict detection can be separated from conflict resolution. When a query and an update are found to conflict, a value-oriented scheduler still has full latitude in deciding how to resolve the conflict. Flexible conflict resolution protocols are easy to use. Different transactions can be aborted or blocked, depending on circumstances. For example, one of each pair of conflicting transactions can be queued and its execution resumed later unless the frequency of deadlock is too high, in which case the least advanced conflicting transaction can be chosen as the one to be aborted and restarted after a random wait.

  Under an item-oriented automatic locking scheme, if locks are granted during the execution of an access command, as items are touched, then it is more difficult to choose dynamically how to resolve conflicts, because partially executed access commands must be rolled back.

- Fourth, scheduling is separated from query execution. If a scheduler detects potential conflicts by analyzing access commands before their execution, it can remain ignorant of how the commands are executed. Conversely, whenever a value-oriented scheduler passes a command to the component of a database system that actually executes commands, that component need not know which transaction the command came from: it can behave as if all access commands that reach it were issued by a single user.

The access routines and data structures of a database system using value-oriented concurrency control can therefore be designed quite separately from the system's scheduler, with only maximal concurrency and efficiency in mind. No further checking for conflict is necessary, and commands can be executed using shared access paths and indexes. It can be efficient sometimes not to flush caches between different transactions, or to read a whole relation from disk once for use by a number of commands. All these optimizations can be implemented, with no effect on the correctness of scheduling.

In summary, it is natural to use a hierarchy of levels of abstraction when specifying what functions a database system should support. Value-oriented concurrency control allows the hierarchy to be respected in the implementation of a system.

2.4 Predicate locking and its problems

The previous section argued that value-oriented concurrency control is important because it should be able to provide increased flexibility in the design of a database system. It is worthwhile to describe previous research on value-oriented concurrency control in order to understand the important difficulties that resulted in the neglect of the idea. Five proposals are discussed in this section and the following one: first the original value-oriented scheme called predicate locking, and then four later schemes called precision locking, expression locking, view locking and predicative optimistic concurrency control.

The seminal paper on concurrency control was written by Jim Gray and others at IBM San Jose Research Center in the mid 1970s [Eswaran *et al.*, 1976]. At the time this paper was written, its authors were working on System R, the first large scale implementation of a relational database system. The early designs for System R used a value-oriented concurrency control scheme called predicate locking, but the idea was abandoned quickly in favour of item-oriented locking [Chamberlin *et al.*, 1981]. The positive reason for adopting item-oriented locking was that the insertion of a phantom tuple can be prevented by locking a region of an index in which it should be entered, as explained above. The negative reasons, however, were more important. Three major difficulties made reasoning about predicate locks appear impossible to do efficiently. These are

(i) inventing expressions describing what parts of a database are retrieved or updated by each access command;

(ii) checking whether these expressions indicate a conflict;

(iii) reducing spurious reports of potential conflict.

The remainder of this section discusses the three difficulties one by one, and the following section evaluates to what extent the successors of predicate locking resolve them.

2.4.1 Inventing read and write set expressions

Under any locking scheme, item-oriented or value-oriented, it is vital to ensure that the locks granted to a transaction do protect all the parts of the database that the transaction accesses. The information retrieved by a command is called its read set, and the information changed by a command is called its write set. A system using a value-oriented locking scheme, where protected extents of data are described by logical expressions, needs logical expressions describing the read and write sets of each transaction. These expressions must be in the same language as those used to describe locked parts of the database.

The read set of a query is not the same as its result, because answering a query often requires accessing tuples not in the result. Inventing descriptions for read and write sets is therefore not trivial. Moreover, expressions describing read sets must be invented before executing a query or an update, since the scheduler must know that the access command is protected before executing it.

It is not obvious how to derive logical expressions describing read and write sets efficiently by analyzing access commands. This difficulty has not been remarked before now, perhaps because it is easy for an item-oriented scheduler to determine which fixed parts of the database an access command refers to or updates. An item-oriented scheduler can also observe what items are touched during the execution of queries and updates.

2.4.2 Checking conflict between expressions

Consider two locks protecting sets of tuples defined by logical expressions. Checking whether the locks conflict means deciding whether the two sets may intersect, that is whether some tuple might exist that satisfies all constraints specified by both expressions. For arbitrary logical expressions, the satisfaction problem is recursively undecidable. The inventors of predicate locking therefore proposed that the constraints allowed in a lock expression should be restricted

(i) to concern attributes of one relation only, and

(ii) to be $<$, $\leq$, $=$, $\geq$, $>$, and $\neq$ comparisons between attributes and/or constants, combined by the standard logical connectives.

An expression of this type is called a *selection formula.* For example, the selection formula

$$\textsf{account}: \, age > 18 \wedge balance > 10000 \wedge income > balance.$$

describes a good customer for a credit card company. Unfortunately, although the problem is not undecidable, it is an $\mathcal{NP}$-complete decision problem to check whether two selection formulas are simultaneously satisfiable [Hunt and Rosenkrantz, 1979].

2.4.3 Reducing spurious reports of potential conflict

Under predicate locking two transactions are prevented from executing concurrently if there *might* exist a tuple that one wants to lock in exclusive mode and the other wants to lock also. Two lock expressions may be simultaneously satisfiable even though no tuple actually satisfying both formulas exists. When this happens, predicate locking prevents transactions from executing that in fact do not conflict. To increase throughput, it is important to eliminate spurious reports of potential conflict as far as possible. In particular, if integrity constraints imply that a tuple cannot exist, it is important to be able to use that fact to infer that two transactions cannot possibly conflict. As an example, recall the query read $\sigma_{C=New\ York}(\mathsf{client}\bowtie\mathsf{office})$ and the update

$$\mathsf{delete\ client}: \ \pi_{PB}\left(\sigma_{C=Los\ Angeles}(\mathsf{client}\bowtie\mathsf{office})\right).$$

If B were not a key of the relation office, that is if more than one C value could match a B value, then a real conflict might exist because the update could cause a change in the answer to the query. It is thus necessary to use the fact that C is a key to avoid spuriously reporting a potential conflict.

The original predicate locking scheme also gave rise to spurious reports of potential conflict for a second reason. Consider an access command involving a join, say read $\mathsf{R}\underset{A=B}{\bowtie}\mathsf{S}$. A tuple only belongs to the read set relative to R of this command if its component value for the attribute A is the same as the value for the attribute B of some tuple of S. No logical formula referring only to attributes of R can express this connection, so a predicate locking scheduler must protect the command read $\mathsf{R}\underset{A=B}{\bowtie}\mathsf{S}$ by locks on the entire relations R and S. The concurrent execution of transactions may be completely inhibited. For a real example of this problem, consider again the query read $\sigma_{C=New\ York}$(client⋈office). No selection formula can exactly describe the read set relative to client of this query.

2.5 The successors of predicate locking

Although mainstream research on database concurrency control has concentrated on item-oriented schemes, a number of attempts to alleviate the problems of predicate locking have been published. Four are discussed in this section; more exist [Wheeler, 1986]. The authors of the schemes described here have not reported successfully implementations of them, and they have not proposed further refinements of their schemes.

2.5.1 Precision locking

The value-oriented locking scheme described in [Jordan *et al.*, 1981] is called "precision locking" by its authors. Under predicate locking, each transaction acquires exclusive and shared mode locks on various parts of the database. Under precision locking, exclusive mode lock expressions are replaced by a special update table where all tuples inserted or deleted by each transaction are recorded until that transaction commits or aborts. Whenever any transaction claims a shared mode lock on some set of tuples, the scheduler applies the expression describing that set to all the tuples recorded in the update table by other active transactions. If any of these tuples satisfies the constraints of the lock expression, then its transaction conflicts with the transaction claiming the lock. A scheduler that uses precision locking never checks whether two lock expressions are simultaneously satisfiable. Instead it checks whether individual concrete tuples satisfy lock expressions.

Precision locking solves the second and third problems of predicate locking listed in the previous section: expressions are never tested for satisfiability, and conflicts are only asserted if they actually occur. However precision locking still suffers from the first problem of predicate locking, that of inventing read set expressions. The inventors of precision locking suggested that shared mode locks should be claimed automatically, but they did not present any method of inventing expressions to describe the read sets of transactions.

Precision locking has a significant new drawback compared to predicate locking. Under predicate locking, it is in principle possible to decide whether a lock request may be granted by analyzing the expression in the request. However under precision locking, write set expressions are replaced by the set of tuples that they describe. An arbitrarily short expression can describe arbitrarily many tuples. Precision locking must deal with these tuples one-by-one. As well as considerable computing effort, this can require extra disk retrieval operations, which is unacceptable because as mentioned before, in typical database systems, disk throughput is the major limiting factor on performance.

2.5.2 Expression locking

The principal idea of the scheme called "expression locking," introduced in [Klug, 1983], is to allow the user to write arbitrary relational algebra expressions in locking commands. Some of the problems of predicate locking are then remedied. In particular, the second cause of spurious reports of potential conflict can be eliminated, because a read or write set expression can mention more than one relation.

The difficulties of expression locking lie in its implementation. The algorithms sketched in [Klug, 1983] (for checking whether lock expressions protect access commands and whether the locks of two transactions conflict) are based on a formal notation incapable of representing $\leq$, $\geq$, or $\neq$ conditions. A relational algebra query expression such as $\sigma_{A\leq B}(\mathrm{e})$ must be rewritten as $\sigma_{A<B}(\mathrm{e}) \cup \sigma_{A=B}(\mathrm{e})$ and in effect represented by two expressions. The size of the set of expressions representing a single relational algebra query may be exponential in the length

of the original query expression.

Chapter 4 below provides a notation for relational algebra queries that avoids exponential explosion in representing $\leq$, $\geq$, and $\neq$ conditions. However expression locking has a second difficulty, which is more fundamental. Checking well-formedness, that is whether a transaction's locks do protect its own read and write sets, amounts to deciding whether the set of tuples defined by one expression is included in that defined by another. This decision problem is strictly harder than the problem of whether two expressions intersect. Even for expressions involving no disjunction, it is $\mathcal{NP}$-hard [Aho *et al.*, 1979]. Therefore expression locking is not a practical concurrency control scheme.

2.5.3 Two further proposals

A scheme that can be understood as an optimistic version of precision locking, called "predicative optimistic concurrency control," is described in [Reimer, 1983]. The idea is that updated tuples and read set expressions are accumulated for all transactions, and each transaction is validated by checking that none of its updated tuples are in any read set of any other concurrent transaction. Unfortunately the paper presenting this scheme assumes a procedural language for access commands in which query expressions cannot involve joins. The problems of predicate locking are thus more evaded than solved.

A concurrency control scheme proposed under the name "view locking" is limited in a similar way. The paper that presents this scheme [Blaauw and Duijvestijn, 1985] describes how to derive read set expressions called "view table predicates" from access commands, and suggests that a scheduler check for conflicts by comparing the result of evaluating these expressions to all inserted and deleted tuples. To this extent, view locking is similar to precision locking. What is different is that the novel relational algebra notation for view table predicates of view locking is incapable of representing selections or projections. View locking therefore also provides only a partial solution to the problems of predicate locking, and it is not a practical concurrency control scheme.

Chapter 3

Independence

Two transactions conflict if an update issued by one may change the result of a query command issued by the other. When it is impossible for an update to change the result of a query, then the query and update are said to be independent. As shown by the example in the introduction, independence is an important notion with applications beyond concurrency control. For example, a remote site in a distributed database system need not be informed of an update when the views stored at the site are independent of the update, because independence implies that the extensions of the views do not have to be recomputed.

This chapter investigates the concept of independence abstractly, while keeping in mind the practical aim of deciding independence for simple queries and updates efficiently. Sections 3.1 and 3.2 formalize deductive databases, setting the stage for Section 3.3 to contain a model-theoretic definition of independence. Basic properties of independence are proved in Sections 3.4 and 3.5. In Section 3.6 proof-theoretic conditions for conjunctive queries and updates to be independent are shown correct, while Section 3.7 provides an inductive proof rule that can be used to that a recursive query is independent of an update.

3.1 Deductive databases

Since relational data definition and query languages were first introduced, there have been efforts to increase their expressiveness. Research on deductive databases constitutes the major effort to generalize relational databases.

Each of the relations making up a relational database is normally stored explicitly. On the other hand, only some of the relations making up a deductive database are stored explicitly, whereas others are defined implicitly. The explicitly stored relations of a deductive database system are called *base* or *extensional*, whereas the implicitly defined relations are called *derived* or *intensional*.

The main formalism used in work on deductive databases has traditionally been first-order logic. Relation names are identified with predicate symbols, and atomic formulas describe tuples. For example, the formula

$$client(reagan, schwab)$$

corresponds to the assertion

$$\langle reagan, schwab \rangle \in \mathsf{client}.$$

A definition of a derived relation must name the relation being defined, and provide a way of computing on demand the extension of the derived relation. The class of definite clauses provides a suitable family of definition expressions.

Definition 3.1: A closed first-order sentence of the form

$$\forall x_1, \ldots, x_n \, A_1 \wedge \cdots \wedge A_k \rightarrow H$$

is a *definite clause* if each of A_1 through A_k and H is a positive atomic formula whose variables are drawn from the set $\{x_1, \ldots x_n\}$. ∎

Definite clauses are simple implications that do not involve negation or disjunction. For the present work, they are required to be range-restricted [Ullman, 1988]. In most work on deductive databases, rules are also required to be function-free. That is, they can only involve constants in addition to universally quantified variables. A function-free set of definite clauses is called a DATALOG program [Ullman, 1988]. The results of this chapter do not depend on the absence of function symbols in rules.

Using the machinery of first-order logic, the definition of a deductive database can be made entirely formal.

Definition 3.2: Given a first-order language $\mathcal{L}$, the corresponding *Herbrand universe* U is the set of all terms of the language. A *Herbrand structure* is an interpretation of $\mathcal{L}$ where each term is mapped to itself, and each predicate of arity n is mapped to a subset of U^n. ■

Most work in logic programming restricts attention to Herbrand interpretations. The ultimate reason for this restriction is that it facilitates establishing connections between model-theoretic and proof-theoretic notions. In standard first-order logic, any set is allowed to be the universe of an interpretation, and theorems must be true of all possible interpretations. Some important theorems of logic programming are not true without the restriction to Herbrand interpretations.

The next definition specifies what a deductive database is as a mathematical object.

Definition 3.3: Let $\mathcal{L}$ be a fixed first-order language, and let U be its Herbrand universe. A *deductive database* is a triple $\langle D, R, IC\rangle$ where D is a Herbrand structure, R is a set of range-restricted definite clauses in $\mathcal{L}$, and IC is any set of sentences in $\mathcal{L}$. ■

Using the terminology of relational database theory, the universe U is the set of all possible attribute values. (In other words, U is the union of all domains from which attribute values may be drawn. To capture the restriction that the values of each attribute may only be drawn from its own domain, we could use a manysorted logic, but none of the results below would differ in any important way.) In a relational database system at any moment each relation has a certain state which is a set of tuples of appropriate length. The structure D is a collection of relation extensions for the predicates of the language $\mathcal{L}$: each extension corresponds to the state of one relation.

The set IC is intended to be a collection of integrity constraints. An integrity constraint is a logical assertion that all valid databases are required to satisfy. The enforcement of integrity constraints is stated formally by the next definition. Illustration 3.1 discusses how the asymmetry between D, a syntactic object, and IC, a semantic object, is necessary for IC to have the semantics one wants.

Definition 3.4: The database $\langle D, R, IC \rangle$ is *valid* if for all sentences α in IC, D is a model of α. ■

The definition of a deductive database as a triple $\langle D, R, IC \rangle$ is mathematically elegant but still realistic, since it does not need to be modified to accommodate special operators that are implemented procedurally. The most common of these operators are comparisons such as $\leq$, $=$, and $\neq$. In a query a comparison operator behaves in all respects like an extensional relation, so it need not be distinguished mathematically from a true extensional relation. It is immaterial that the notional extension of a special operator, viewed as a relation, may be infinite, and that the relation is in fact computed by procedural attachment.

In reasoning about queries and updates, we will be making the assumption that the extensions of extensional relations are unavailable. Nevertheless, knowledge about any special relation can be exploited by stating it in the background collection of integrity constraints. Indeed, note that it is not necessary that integrity constraints be represented as logical sentences: they can be built into a specialized decision procedure.

Given a collection of rules R and an extensional database D, the minimal Herbrand model of R that includes D is uniquely defined. From now on $D, R \models \alpha$

A deductive database is a triple $\langle D, R, IC \rangle$ where $D \models IC$: the relation extensions in D are required to satisfy the constraints in IC. It is important to note that D is a semantic object, whereas IC is a syntactic object. Integrity constraints talk about what is in a database as opposed to what is true in the world modelled by the database. If a database contains relation extensions, then it is possible to state properties of those extensions by means of first-order sentences. On the other hand, if a database contains sentences, then to talk about those sentences it is necessary to use a higher-order logic. A suitable logic of knowledge is provided in [Reiter, 1988], but we prefer to formalize extensional databases as logical structures, and to take integrity constraints as just first-order sentences.

As an example, remember the binary relation client with schema PB and the functional dependency $P \rightarrow B$. In the language of deductive databases, *client* is a predicate symbol of arity two. The functional dependency can be expressed as the first-order sentence

$$\forall p, c_1, c_2 \; client(p, c_1) \wedge client(p, c_2) \rightarrow c_1 = c_2.$$

A Herbrand structure will satisfy this sentence if and only if the relation interpreting *client* satisfies the functional dependency.

Illustration 3.1: Integrity constraints–syntax versus semantics.

will mean that α is true in this minimal model. It is important to admit updates that introduce new constant symbols, so we assume that the Herbrand universe contains constants not mentioned in D or R. (Another reason for making this assumption is that it eliminates universal query anomalies [Ross, 1989].)

3.2 Queries and updates

In Section 2.1, query and update commands were discussed with no detail on the syntax of query expressions or their evaluation. This section supplies that detail for queries and updates against deductive database systems as defined above.

A query is an open logical formula written $\phi(\bar{x})$ where $\bar{x}$ is the set of free variables of the query. Relative to a database D and a set of rules R, a query $\phi(\bar{x})$ defines a set of tuples, called the result of the query and written

$$[\![\phi(\bar{x})]\!]_{D,R} = \{\bar{c} \in U^n : D, R \models \phi(\bar{c})\}$$

where n is the number of free variables in the query and U is the Herbrand universe. Note that because rules are definite clauses, the Henkin property holds: if $D, R \models \exists\bar{x}\ \phi(\bar{x})$ then $D, R \models \phi(\bar{c})$ for some tuple $\bar{c}$ of constants in the Herbrand universe.

The definition of the result of a query calls for forming n-tuples of elements of the universe U. A particular D and R never fully determine a universe. All that one can say by examining D and R is that certain constant symbols must belong to R, but it is impossible to say what additional symbols also belong to R. As mentioned before, it is important that U should contain additional symbols, to allow updates to refer to new objects in the outside world. However it would be anomalous for a query to be able to reveal what these additional symbols are. A query is said to be *domain-independent* if its result is unaffected by any change in the universe U: whatever the additional symbols in U, and in particular even if there are no additional symbols, the result of the query should be the same. Domain-independence is an undecidable property of queries, but sufficient syntactic conditions that can be checked quickly are known [Van Gelder and Topor, 1987]. We assume that one of these conditions is imposed on all queries. For example, the query $\phi(x) = \exists y\ P(x,y) \wedge \forall z\ \neg Q(y,z)$ is recognizably domain-independent: while the conjunct $\forall z\ \neg Q(y,z)$ intuitively calls for y values that are not Q-related to any z, the conjunct $P(x,y)$ provides an effective means of generating candidate y values.

We consider a basic class of updates: additions and removals of sets of tuples from extensional relations. Such updates can be written $\mathsf{insert}\ P(\bar{x}) : \phi(\bar{x})$ or $\mathsf{delete}\ P(\bar{x}) : \phi(\bar{x})$ where P names an extensional relation and $\phi(\bar{x})$ is a query. The atoms added or removed by an update are the set $\{P(\bar{x}) : \bar{x} \in [\![\phi(\bar{x})]\!]_{D,R}\}$, which in general depends on the current state of the database. (For an addition $\mathsf{insert}\ P(\bar{x}) : \phi(\bar{x})$ the query $\phi(\bar{x})$ must be domain-independent but for a deletion $\mathsf{delete}\ P(\bar{x}) : \phi(\bar{x})$ it is only necessary that $P(\bar{x}) \wedge \phi(\bar{x})$ be domain-independent.)

3.3 The formal definition of independence

When an update cannot change the result of a query, the query is called *independent* of the update. Our aim is to develop characterizations of independence that can be checked without accessing the extensional database. We do not want to follow precision locking and similar schemes, and fall into the trap of requiring extra database accesses to perform independence reasoning. Given a query and an update, it should be possible to decide their independence without accessing any extensional relation. An arbitrarily short update or query command can concern arbitrarily many tuples, and it is important not to have to deal with the tuples one-by-one.

The following formal definitions therefore involve quantification over all valid extensional databases. The rules of the game are typically similar for query optimization: a query must be reformulated and an evaluation strategy must be

Suppose a database has two binary relations *client* and *office* where $client(p, b)$ signifies that person p is a customer of broker b and $office(b, c)$ signifies that an office of b is located in city c. The query

$$\textsf{read}\ \exists b, c\ client(p, b) \wedge office(b, c) \wedge c = New\ York$$

asks for all persons who have a broker with an office in New York. In relational algebra this query would be written $\pi_P\left(\sigma_{C=New\ York}(\text{client} \bowtie \text{office})\right)$. If the relation *office* is extensional, then

$$\textsf{insert}\ office(b, c) : b = Schwab \wedge c = Los\ Angeles$$

is an update that calls for adding the tuple $\langle Schwab, Los\ Angeles\rangle$ to *office*. Note that this update does not depend on the current state of the database, but it may be illegal if the database already contains a fact such as $office(Schwab, Chicago)$ and the functional dependency $office : b \rightarrow c$ is imposed.

Illustration 3.2: An example of a query and an update.

chosen without evaluating the query in question.

Definition 3.5: Let IC be a background set of integrity constraints and let R be a background set of rules. The query $\alpha = \phi(\bar{x})$ is *independent* of the update $\mathsf{insert}\ Q(\bar{y}) : \psi(\bar{y})$ if for all extensional databases D satisfying IC, if $D \cup \Delta$ satisfies IC then

$$[\![\alpha]\!]_{D,R} = [\![\alpha]\!]_{D \cup \Delta, R}$$

where $\Delta = \{Q(\bar{y}) : \bar{y} \in [\![\psi(\bar{y})]\!]_{D,R}\}$. ■

Intuitively, $D \cup \Delta$ is the result of adding to the extensional database the set of tuples specified by the update. If the set of tuples defined by a query remains the same, then the query is independent of the update.

The definition of independence for deletion updates is similar. One of the results of the next section addresses the question of when an independence fact about one type of update can be extended to the other type.

Definition 3.6: Let IC be a background set of integrity constraints and let R be a background set of rules. The query $\alpha = \phi(\bar{x})$ is *independent* of the update $\mathsf{delete}\ Q(\bar{y}) : \psi(\bar{y})$ if for all extensional databases D satisfying IC, if $D \setminus \Delta$ satisfies IC then

$$[\![\alpha]\!]_{D \setminus \Delta, R} = [\![\alpha]\!]_{D,R}. \ ■$$

Again, intuitively $D \setminus \Delta$ is the result of removing the set of tuples specified by the update.

In the rest of this chapter the background set of rules R is always arbitrary but fixed. The background set of integrity constraints IC is also always fixed, but sometimes it is required to satisfy certain conditions.

3.4 Basic properties of independence

Our final aim is to find algorithms for deciding quickly whether a query is independent of an update. This section contains some preliminary lemmas.

The obvious first question is whether independence is decidable. The following reduction is due to Alberto Mendelzon [personal communication].

Lemma 3.1: Independence is undecidable.

Proof: It is a known result of DATALOG theory that given rules defining two predicates P_1 and P_2, it is undecidable whether an intensional relation P_1 is a subrelation of another intensional relation P_2 whatever the extensional database [Shmueli, 1987]. Construct a deductive database $\langle D, R, IC\rangle$ as follows. Let IC be empty. Let B be a new extensional relation and let the update U be insert $B(\bar{x}) : P_1(\bar{x})$. Let R be the rules for P_1 and P_2 and

$$\{Q(\bar{x}) \leftarrow B(\bar{x}), Q(\bar{x}) \leftarrow P_2(\bar{x})\}.$$

Now the query $Q(\bar{x})$ is independent of U if and only if P_1 is always a subrelation of P_2. ∎

The definitions of independence become vacuous if D is not valid or if, respectively, $D \cup \Delta$ or $D \setminus \Delta$ is not valid. Intuitively, updates that do not preserve validity are simply excluded from consideration. To enable conclusions to be drawn about independence given only that D is valid, a condition guaranteeing validity preservation is needed.

Definition 3.7: The set of integrity constraints IC is *downward-hereditary* if $D \models IC$ implies $D' \models IC$, for all extensional databases D' and D such that $D' \subseteq D$. ∎

Every set of functional dependencies is downward-hereditary. Indeed, every collection of equality-generating dependencies and monotonic dependencies [Brodsky and Sagiv, 1989] is downward-hereditary. Tuple-generating dependencies [Maier, 1983] are not downward-hereditary, but they can be replaced by rules by introducing new intensional relations.

Definition 3.8: The query α is *monotonic* if $[\![\alpha]\!]_{D',R} \subseteq [\![\alpha]\!]_{D,R}$ for all extensional databases D' and D such that $D' \subseteq D$. The query β is *anti-monotonic* if $\neg\beta$ is monotonic. ∎

Monotonicity is the condition that the answer to a query should expand if the extensional database expands. Typical queries asking for "positive" information are monotonic. All DATALOG queries are monotonic. The next result says that independence of a deletion update is a more stringent condition than independence of an addition update.

Lemma 3.2: Let $\langle D, R, IC\rangle$ be a deductive database where IC is a downward-hereditary set of integrity constraints, and let α and $\phi(\bar{x})$ be monotonic queries. If α is independent of the update $\mathsf{delete}\ P(\bar{x}) : \phi(\bar{x})$ then α is also independent of $\mathsf{insert}\ P(\bar{x}) : \phi(\bar{x})$.

Proof: Assume α is independent of $\mathsf{delete}\ P(\bar{x}) : \phi(\bar{x})$. Suppose both D and $D \cup \Delta$ satisfy IC, where $\Delta = \{P(\bar{x}) : \bar{x} \in [\![\phi(\bar{x})]\!]_{D,R}\}$. We must show that $[\![\alpha]\!]_{D,R} = [\![\alpha]\!]_{D\cup\Delta,R}$.

Consider $D' = D \cup \Delta - \{P(\bar{x}) : \bar{x} \in [\![\phi(\bar{x})]\!]_{D\cup\Delta,R}\}$. By the monotonicity of $\phi(\bar{x})$, $D' \subseteq D$. The downward closure of IC implies that D' satisfies IC. So $[\![\alpha]\!]_{D',R} = [\![\alpha]\!]_{D\cup\Delta,R}$ because α is independent of $\mathsf{delete}\ P(\bar{x}) : \phi(\bar{x})$. But $[\![\alpha]\!]_{D',R} \subseteq [\![\alpha]\!]_{D,R} \subseteq [\![\alpha]\!]_{D\cup\Delta,R}$ by the monotonicity of α. It follows that $[\![\alpha]\!]_{D,R} = [\![\alpha]\!]_{D\cup\Delta,R}$. ∎

The claim of Lemma 3.2 also holds if the query α is anti-monotonic.

3.5 Independence of component queries

Lemma 3.3: Let IC be a downward-hereditary set of integrity constraints. Let $U_1 = \mathsf{delete}\ Q(\bar{x}) : \phi_1(\bar{x})$ and $U_2 = \mathsf{delete}\ Q(\bar{x}) : \phi_2(\bar{x})$ be two updates such that $[\![\phi_1(\bar{x})]\!]_{D,R} \subseteq [\![\phi_2(\bar{x})]\!]_{D,R}$ for all valid D. Then for all monotonic queries α, α independent of U_2 implies α independent of U_1.

Proof: Suppose α is independent of U_2 and the extensional databases D and $D \setminus \Delta$ satisfy IC where $\Delta = \{P(\bar{x}) : \bar{x} \in [\![\phi_1(\bar{x})]\!]_{D,R}\}$. We must show that

$[\![\alpha]\!]_{D,R} = [\![\alpha]\!]_{D\setminus\Delta,R}$. Consider $\Delta' = \{P(\bar{x}) : \bar{x} \in [\![\phi_2(\bar{x})]\!]_{D,R})\}$. By downward closure $D \setminus \Delta'$ satisfies IC so $[\![\alpha]\!]_{D,R} = [\![\alpha]\!]_{D-\Delta',R}$. Now $\Delta \subseteq \Delta'$ so by the monotonicity of α, $[\![\alpha]\!]_{D\setminus\Delta',R} \subseteq [\![\alpha]\!]_{D\setminus\Delta,R} \subseteq [\![\alpha]\!]_{D,R}$ and the result follows. ■

Again, the previous claim also holds for anti-monotonic queries. The example of Illustration 3.3 shows that no similar result holds for pairs of queries α_1 and α_2 such that $[\![\alpha_1]\!]_{D,R} \subseteq [\![\alpha_2]\!]_{D,R}$ always.

Imagine a database of stockmarket information containing relations named *underpriced*, *leveragedbuyout*, and *stockclass*, where *underpriced*(s) signifies that the security s is priced relatively low, *leveragedbuyout*(s) means that the company issuing s is going private, and *stockclass*(s, c) means that s is of type c. Consider two queries to find stocks worth purchasing:

$$\textsf{read } \phi_1(x) = \mathit{underpriced}(x) \wedge \mathit{stockclass}(x, \mathit{preferred})$$

$$\textsf{read } \phi_2(x) = \mathit{underpriced}(x).$$

Clearly for all D and R the answer to the first query is contained in the answer to the second: $[\![\phi_1(x)]\!]_{D,R} \subseteq [\![\phi_2(x)]\!]_{D,R}$. Now consider two updates to eliminate stocks that should be dropped from the database:

$$U_1 = \textsf{delete } \mathit{underpriced}(x) : \mathit{stockclass}(x, \mathit{nonvoting})$$

$$U_2 = \textsf{delete } \mathit{stockclass}(x, y) : \mathit{leveragedbuyout}(x).$$

Given a functional dependency of the second attribute of *stockclass* on the first attribute, the query $\phi_1(x)$ is independent of U_1 but not of U_2, whereas $\phi_2(x)$ is independent of U_2 but not of U_1.

Illustration 3.3: Independence versus subsumption.

The following lemma allows one to prove that a query is independent of an update by showing that each logical component of the query is independent.

Lemma 3.4: Let $\phi(\bar{x})$ and $\psi(\bar{x})$ be queries over the same free variables, and let U be an update. If $\phi(\bar{x})$ and $\psi(\bar{x})$ are independent of U, then so are the queries $\phi(\bar{x}) \wedge \psi(\bar{x})$, $\phi(\bar{x}) \wedge \neg\psi(\bar{x})$, $\phi(\bar{x}) \vee \psi(\bar{x})$, and $\exists \bar{x}' \, \phi(\bar{x})$ where $\bar{x}'$ is a subset of the set of variables $\bar{x}$. ∎

Unfortunately the converse of Lemma 3.4 does not hold: a query can be independent of an update even though some of its component queries are not independent of the update. Illustration 3.3 shows that it is possible for the query $\phi(\bar{x}) \wedge \psi(\bar{x})$ to be independent of an update when neither or only one of its two subqueries is independent. Less obviously, the same is true of $\phi(\bar{x}) \vee \psi(\bar{x})$, even when neither subquery is subsumed by the other, or unsatisfiable. Thus we do not have a normal form for queries that would enable Lemmas 3.2 and 3.3 to be extended to all queries by considering their monotonic and anti-monotonic elementary components.

3.6 Independence for conjunctive queries

This section provides a characterization of independence that applies when an update concerns a relation that is only involved in a query directly: that is, the relation is not mentioned in any rules defining any intensional relation used in the query. There is a special intermediate case between full deductive databases and standard relational database systems, where nonrecursive rules are allowed, but not recursive rules. A derived relation defined by a nonrecursive rule is often called a view. Since unfolding nonrecursive rules is a process that always terminates, the results of this section apply to all queries and updates in traditional relational database systems, including when queries use views.

The next definition and lemma deal with the case where a query is trivially independent of an update.

Definition 3.9: Let M_0 be the relations mentioned explicitly in a query α, and let M_{i+1} be the relations mentioned in the rules defining the intensional relations in M_i. The relations *used indirectly* in α are the set $M(\alpha) = \bigcup_{i \geq 0} M_i$. ■

The following claim is obvious. We assume here and in the remainder of this thesis that any symbolic expression is available in parse tree form. The length $|\alpha|$ of an expression α is the number of nodes in its parse tree. The expressions we consider always follow very simple grammars, and their text string representations can be parsed in time linear in the number of characters of the string.

Lemma 3.5: For any set of rules R and query α, it is possible to compute $M(\alpha)$ in $O(|R| + |\alpha|)$ time, and if $Q \notin M(\alpha)$ then α is independent of any update $U = \pm Q(\bar{y}) : \psi(\bar{y})$. ■

Our general aim is to find efficiently decidable sufficient conditions for when a query is independent of an update. The definition of independence is model-theoretic, but these conditions will be proof-theoretic. The next definition specifies the class of queries to which the conditions of this section apply.

Definition 3.10: The query $\phi(\bar{x})$ is *conjunctive in* Q if it is of the form $\exists \bar{z}\ Q(\bar{x}', \bar{z}') \wedge \phi'(\bar{x}, \bar{z})$ where $Q(\bar{x}', \bar{z}')$ is a positive literal with arguments $(\bar{x}', \bar{z}')$ taken from the free variables $\bar{x}$ and the existentially quantified variables $\bar{z}$, and $Q \notin M\left(\phi'(\bar{x}, \bar{z})\right)$. ■

What is conventionally called a conjunctive query [Ullman, 1982] is conjunctive according to Definition 3.10 in each relation that it mentions. Such a query can be called *fully* conjunctive. The results of previous work on conjunctive queries apply only to fully conjunctive queries, so the results below are more general in that they apply to queries that are only conjunctive in some relations.

The following theorems reduce independence to unsatisfiability when a query is conjunctive in the relation affected by an update. The unsatisfiability conditions are stated using a provability relation $\vdash$ as $IC, R \vdash \Psi$. This provability relation is intended to correspond to an actual decision procedure, so it is only required to be sound, not necessarily complete. Soundness means that if $IC, R \vdash \Psi$ and $D \models IC$, then $D, R \models \Psi$.

Theorem 3.6: Let IC be a background set of integrity constraints. The query $\phi(\bar{x})$ is independent of the update $U = \textsf{delete}\ Q(\bar{y}) : \psi(\bar{y})$ if it is conjunctive in Q and

$$IC, R \vdash \nexists \bar{x}, \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}') \wedge \psi(\bar{x}', \bar{z}'),$$

where $\phi(\bar{x}) = \exists \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}')$.

Proof: Suppose the unsatisfiability condition holds but $\phi(\bar{x})$ is not independent of U. For some D and $\bar{c}$, it must be the case that $\bar{c} \in [\![\phi(\bar{x})]\!]_{D,R} \setminus [\![\phi(\bar{x})]\!]_{D^-,R}$ where $D^- = D \setminus \{Q(\bar{y}) : \bar{y} \in [\![\psi(\bar{y})]\!]_{D,R}\}$. We shall derive a contradiction.

We have that $D, R \models \phi(\bar{c})$ but $D^-, R \not\models \phi(\bar{c})$. Write $\phi(\bar{c})$ as $\exists \bar{z}\ \phi'(\bar{c}, \bar{z}) \wedge Q(\bar{c}', \bar{z}')$. It is the case that $D, R \models \exists \bar{z}\ \phi'(\bar{c}, \bar{z})$ if and only if $D^-, R \models \exists \bar{z}\ \phi'(\bar{c}, \bar{z})$, because Q is not mentioned or used indirectly in $\phi'(\bar{x}, \bar{z})$, and D differs from D^- in Q facts only. Therefore there must be some $\bar{d}'$ such that $D, R \models Q(\bar{c}', \bar{d}')$ but $D^-, R \not\models Q(\bar{c}', \bar{d}')$. Now $Q(\bar{c}', \bar{d}') \in \{Q(\bar{y}) : \bar{y} \in [\![\psi(\bar{y})]\!]_{D,R}\}$ so $D, R \models \psi(\bar{c}', \bar{d}')$. Now we have the contradiction: $D, R \models \phi'(\bar{c}, \bar{d}) \wedge Q(\bar{c}', \bar{d}') \wedge \psi(\bar{c}', \bar{d}')$ but $D \models IC$ also, and $IC, R \vdash \nexists \bar{x}, \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}') \wedge \psi(\bar{x}', \bar{z}')$. ∎

The next theorem and its proof are very similar, but the extra condition that the update must be monotonic is needed.

Theorem 3.7: Let IC be a background set of integrity constraints. A query $\phi(\bar{x}) = \exists \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}')$ that is conjunctive in Q is independent of the monotonic update $U = \textsf{insert}\ Q(\bar{y}) : \psi(\bar{y})$ if

$$IC, R \vdash \nexists \bar{x}, \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}') \wedge \psi(\bar{x}', \bar{z}').$$

Proof: Suppose the unsatisfiability condition holds but $\phi(\bar{x})$ is not independent of U. For some D and $\bar{c}$, it must be the case that $\bar{c} \in [\![\phi(\bar{x})]\!]_{D^+,R} \setminus [\![\phi(\bar{x})]\!]_{D,R}$, where $D^+ = D \cup \{Q(\bar{y}) : \bar{y} \in [\![\psi(\bar{y})]\!]_{D,R}\}$.

We have that $D^+, R \models \phi(\bar{c})$ but $D, R \not\models \phi(\bar{c})$. Write $\phi(\bar{c})$ as $\exists \bar{z}\ \phi'(\bar{c}, \bar{z}) \wedge Q(\bar{c}', \bar{z}')$. It must be for some $\bar{d}$ that $D^+, R \models Q(\bar{c}', \bar{d}')$, because Q is not used in $\phi'(\bar{x}, \bar{z})$, but $D, R \not\models Q(\bar{c}', \bar{d}')$. Now $Q(\bar{c}', \bar{d}') \in \{Q(\bar{y}) : \bar{y} \in [\![\psi(\bar{y})]\!]_{D,R}\}$ so $D, R \models \psi(\bar{c}', \bar{d}')$. Remember that $\psi(\bar{x}', \bar{z}')$ is monotonic so $D^+, R \models \psi(\bar{c}', \bar{d}')$ also. Now a contradiction is apparent: $D^+, R \models \phi'(\bar{c}, \bar{d}) \wedge Q(\bar{c}', \bar{d}') \wedge \psi(\bar{c}', \bar{d}')$ but $D^+ \models IC$ also, and $IC, R \vdash \nexists \bar{x}, \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}') \wedge \psi(\bar{x}', \bar{z}')$. ∎

Consider the conjunctive query

$$\textsf{read}\ \exists c\ \mathit{office}(b, c) \wedge \mathit{rainy}(c)$$

and the update

$$\textsf{insert}\ \mathit{office}(b, c) : b = \mathit{Schwab} \wedge c = \mathit{Los\ Angeles}.$$

Theorem 3.7 confirms the intuition that the query and the update are independent if conjoining the atomic formulas used to define them gives an unsatisfiable sentence: that is, if

$$IC, R \vdash \nexists b, c\ \mathit{office}(b, c) \wedge \mathit{rainy}(c) \wedge b = \mathit{Schwab} \wedge c = \mathit{Los\ Angeles}.$$

Even a limited decision procedure can presumably perform this inference.

Illustration 3.4: Independence reduced to unsatisfiability.

The unsatisfiability conditions of Theorems 3.6 and 3.7 are necessary as well as sufficient if the provability relation $\vdash$ is complete.

Theorem 3.8: Let $\vdash$ be sound and complete relative to $\models$. If the background set of integrity constraints IC is downward-hereditary, the query $\phi(\bar{x}) = \exists \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}')$ is conjunctive in Q, and

$$IC, R \not\vdash \nexists \bar{x}, \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}') \wedge \psi(\bar{x}', \bar{z}'),$$

then $\phi(\bar{x})$ is not independent of the update $U = \textsf{delete } Q(\bar{y}) : \psi(\bar{y})$.

Proof: Let Ψ be the sentence $\nexists \bar{x}, \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}') \wedge \psi(\bar{x}', \bar{z}')$. Suppose $IC, R \nvdash \Psi$. By the completeness of $\vdash$, $IC, R \not\models \Psi$. That means there exists a D such that $D \models IC$ and $D, R \models \phi'(\bar{c}, \bar{d}) \wedge Q(\bar{c}', \bar{d}') \wedge \psi(\bar{c}', \bar{d}')$, because of the Henkin property that existential witnesses exist. Clearly $\bar{c} \in [\![\phi(\bar{x})]\!]_{D,R}$ and $(\bar{c}', \bar{d}') \in [\![\psi(\bar{y})]\!]_{D,R}$ so $Q(\bar{c}', \bar{d}') \in [\![\psi(\bar{y})]\!]_{D,R}$ so $\bar{c} \notin [\![\phi(\bar{x})]\!]_{D \setminus \{Q(\bar{y}) : \bar{y} \in [\![\psi(\bar{y})]\!]_{D,R}, R}}$. ∎

The converse of Theorem 3.7 is proved similarly.

Conditions of the form $IC, R \vdash \nexists \bar{x}, \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}') \wedge \psi(\bar{x}', \bar{z})$ can sometimes be decided very efficiently. When all rules are unfolded, the query and the update are fully conjunctive, and IC consists of functional dependencies and the axioms for a dense partial order, then the decision procedure presented in the next chapters typically requires linear time, and in the worst case quadratic time.

3.7 Independence for recursive queries

In this section we consider the general case where an update concerns a relation that is used indirectly in a query, through recursive rules. Illustration 3.5 demonstrates a strategy for proving a recursive query independent of an update. The strategy is proved correct in the theorem that follows.

The recursion used in Illustration 3.5 to define the *buys* relation happens to be bounded and linear [Naughton and Sagiv, 1987], but the proof technique works for unbounded nonlinear recursions also. Of course, independence is undecidable so there are true independence assertions that cannot be proved using the theorem, even some involving only one bounded linear recursion.

The next theorem is about the provability of independence facts, so it is necessary to explain what provable means. We already have formal entailment ($\models$) and provability ($\vdash$) relations. These can be extended to 'is independent of'

Consider two recursive rules defining an intensional relation *buys* in terms of two base relations *likes* and *trendy*:

$$buys(x,y) \leftarrow likes(x,y)$$

$$buys(x,y) \leftarrow trendy(x) \wedge trendy(w) \wedge buys(w,y).$$

Is the query **read** $buys(x,y) \wedge trendy(x)$ independent of the update $U =$ **delete** $likes(x,y) : \neg trendy(x)$? The answer is yes, and it can be proved by an inductive method. Take the hypothesis

$$H_0 : buys(x,y) \wedge trendy(x) \text{ is independent of } U.$$

We must prove that the two subqueries that the query gives rise to are independent of the update, using H_0 if we want. (Variable names must be standardized apart as appropriate.) Theorem 3.6 implies that the base case subquery $likes(x,y) \wedge trendy(x)$ is independent of U. Consider the other subquery, which is $\big(\exists w\ trendy(x) \wedge trendy(w) \wedge buys(w,y)\big) \wedge trendy(x)$. By Lemma 3.5 $trendy(x)$ is independent of U. By Lemma 3.4, H_0 implies that $\exists w\ trendy(w) \wedge buys(w,y)$ is independent of U. The proof is now finished.

Illustration 3.5: Independence for a recursive query.

assertions. Definitions 3.5 and 3.6 give a meaning to the statement

$$IC, R \models \alpha \text{ is independent of } U.$$

The lemmas and theorems of Sections 3.4, 3.5, and 3.6 provide ways of proving 'is independent of' assertions. For example, Theorem 3.6 justifies the inference rule

$$\frac{\not\exists \bar{x}, \bar{z}\ \phi'(\bar{x},\bar{z}) \wedge Q(\bar{x}',\bar{z}') \wedge \psi(\bar{x}',\bar{z}')}{\phi'(\bar{x},\bar{z}) \wedge Q(\bar{x}',\bar{z}') \text{ is independent of } \mathsf{delete}\ Q(\bar{y}) : \psi(\bar{y})}$$

with the side-condition

$$Q \notin M\left(\phi'(\bar{x},\bar{z})\right).$$

The next theorem provides a further, inductive, inference rule.

Theorem 3.9: Let IC be a set of integrity constraints, and let R be a set of rules. Let U be an update and let $\mathsf{read}\ \phi(\bar{x}) \wedge P(\bar{x})$ be a query where P is an intensional relation. Without loss of generality, suppose the rules in R defining P are rectified so they can be written $P(\bar{x}) \leftarrow B_j(\bar{x})$ for $j = 1, \ldots, k$. These rules may be recursive, so P is mentioned in $B_j(\bar{x})$, but assume P is not used indirectly.

Consider the hypothesis

$$H : \phi(\bar{x}) \wedge P(\bar{x}) \text{ is independent of } U.$$

If for each $j = 1, \ldots, k$

$$IC, R, H \vdash \phi(\bar{x}) \wedge B_j(\bar{x}) \text{ is independent of } U$$

then $IC, R \models \phi(\bar{x}) \wedge P(\bar{x})$ is independent of U.

Proof: Let $P^0, P^1, \cdots$ be new intensional relation symbols. Consider the collection of rules R' formed by adding to R the rules

$$P^0(\bar{x}) \leftarrow B_1^{-1}(\bar{x})$$

$$\vdots$$

$$\begin{array}{rcl} P^0(\bar{x}) & \leftarrow & B_k^{-1}(\bar{x}) \\ & \vdots & \\ P^{n+1}(\bar{x}) & \leftarrow & B_1^n(\bar{x}) \\ & \vdots & \\ P^{n+1}(\bar{x}) & \leftarrow & B_k^n(\bar{x}) \\ & \vdots & \end{array}$$

where $B_j^n(\bar{x})$ is $B_j(\bar{x})$ with P^n substituted for P, except $B_j^{-1}(\bar{x})$ is $B_j(\bar{x})$ if P does not appear in it, *false* otherwise. Clearly for any extensional database D

$$[\![\phi(\bar{x}) \wedge P(\bar{x})]\!]_{D,R} = \bigcup_{n\in\omega} [\![\phi(\bar{x}) \wedge P^n(\bar{x})]\!]_{D,R'}.$$

Thus to prove that $\phi(\bar{x}) \wedge P(\bar{x})$ is independent of U, it is sufficient to prove that $\phi(\bar{x}) \wedge P^n(\bar{x})$ is independent of U for all $n \in \omega$. We can use induction: it is sufficient to prove for all n that $\phi(\bar{x}) \wedge P^n(\bar{x})$ is independent of U assuming that $\phi(\bar{x}) \wedge P^m(\bar{x})$ is independent of U for $m < n$.

Consider a particular n. The clauses for P^n say that it is enough to show that $\phi(\bar{x}) \wedge B_j^{n-1}(\bar{x})$ is independent of U for $j = 1, \ldots, k$. Let Γ be any proof that $\phi(\bar{x}) \wedge B_j(\bar{x})$ is independent of U, possibly using the hypothesis H. We shall show that Γ can be mapped to a proof Γ' that $\phi(\bar{x}) \wedge B_j^{n-1}(\bar{x})$ is independent of U, if necessary using the hypothesis $H' : \phi(\bar{x}) \wedge P^{n-1}(\bar{x})$ is independent of U.

$B_j(\bar{x})$ can be written $L_1 \wedge \cdots \wedge L_q \wedge \psi$ where each L_p is a P literal and P is not mentioned or used indirectly in ψ. The only way to prove that $\phi(\bar{x}) \wedge B_j(\bar{x})$ is independent of U using the results of Sections 3.4, 3.5, and 3.6 is to use Lemma 3.4 after showing that $L_p \wedge \psi$ is independent of U for each p. Note that $L_p \wedge \psi$ is conjunctive in P, so without loss of generality let $q = 1$.

Let the 'basic' independence assertions of Γ be those whose subproof does not involve any other independence assertions. Let Ψ be a basic independence assertion, say "α is independent of V". If α does not use P then let Ψ and its subproof appear in Γ'. Otherwise, there are only four ways Ψ could be proved: using the hypothesis H, Lemma 3.5, Theorem 3.6, or Theorem 3.7. We shall consider them one by one. Let α' be α with P replaced by P^{n-1} and let Ψ' be $IC, R' \models \alpha'$ is independent of V.

(i) Ψ is in fact H. Clearly Ψ' is H'.

(ii) Ψ follows from Lemma 3.5. $M(\alpha)$ and $M(\alpha')$ are identical when restricted to extensional relations, so Ψ' also follows from Lemma 3.5.

(iii) Ψ follows from Theorem 3.6. If $IC, R \vdash \nexists \bar{x}, \bar{z}\ \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}') \wedge \psi(\bar{x}', \bar{z}')$ and $\alpha = \phi'(\bar{x}, \bar{z}) \wedge Q(\bar{x}', \bar{z}')$ is conjunctive in P then $IC, R' \models \nexists \bar{x}, \bar{z}\ \alpha' \wedge Q(\bar{x}', \bar{z}') \wedge \psi(\bar{x}', \bar{z}')$ since $[\![P^n(\bar{x})]\!]_{D,R'} \subseteq [\![P(\bar{x})]\!]_{D,R'}$ for all D. So Ψ' also follows from Theorem 3.6.

(iv) Ψ follows from Theorem 3.7 in a similar fashion to case (iii). ∎

We conjecture that Theorem 3.9 can be proved without enumerating all basic ways of proving independence assertions. By rewriting the rule set in a simple way, the theorem can be extended to apply to queries using a relation defined using mutual recursion.

The method of proof suggested by the theorem is analogous to the method presented in [Elkan and McAllester, 1988]. It is also similar to the method used in [Sagiv, 1987] for proving that a logic program satisfies a tuple-generating dependency, which is not explicitly inductive.

The most powerful tools for analyzing DATALOG program properties have not been used in this chapter, for example methods using automata theory [Vardi, 1989]. Those methods may perhaps give more refined characterizations of independence; in particular, automata theory should enable one to say more about independence for function-free queries. The methods of this chapter do have merit in that they suggest practical algorithms for checking independence.

Chapter 4

A decision procedure for conjunctive query emptiness

This chapter presents a decision procedure for the question of whether a conjunctive query is unsatisfiable. Queries are expressed here in the language of relational algebra, rather than the language of predicate calculus as before. The question is then whether the set of tuples described by a conjunctive relational algebra expression is always empty, whatever the actual instances of the base relations to which the expression refers.

The decision procedure is of practical relevance because it takes into account the constraint that each base relation must satisfy specified functional dependencies. It thus produces answers in polynomial time that are precise enough to be relied upon.

Section 4.1 below discusses conjunctive queries, defines the emptiness problem, and shows that it is $\mathcal{NP}$-hard for queries that are not conjunctive. Section 4.2 describes a new tableau notation for conjunctive queries used by the decision procedure. Section 4.3 presents three conditional rewrite rules that constitute the decision procedure. The next chapter shows how to implement these rules efficiently.

4.1 Conjunctive relational algebra queries

It is a common phenomenon in automated reasoning that computational complexity is caused by the presence of disjunctions. Intuitively, reasoning about a disjunction must be done by analyzing each alternative case separately, and the space of cases to consider is the cross-product of the cases for each disjunction. The total number of cases can therefore be exponential in the number of disjunctions. Reasoning with conjunctions, on the other hand, can be done by accumulating inferences monotonically. Showing exactly how to do this for conjunctive relational algebra queries is the topic of this chapter.

Definition 4.1: A *conjunctive query* is any relational algebra expression containing only join, projection, intersection, and selection operators, where all selection conditions are conjunctions of $<$, $\leq$, $=$, $\geq$, $>$, or $\neq$ comparisons between an attribute variable and a constant or another attribute variable. ■

Conjunctive queries are also known as select-project-join expressions. An example of the type of selection condition allowed in a conjunctive query is

$$salary > balance$$

and an example of a conjunctive query using this condition is

$$\sigma_{(salary > balance) \wedge (balance > 10000)}(\textsf{account}).$$

The conjunctive queries defined here are more general than those defined elsewhere. Unlike in [Klug, 1983], all six comparison operators are allowed in selection conditions. A consequence of this latitude is that the negation of any atomic selection condition, say $salary > balance$, is also an atomic selection condition, in this case $salary \leq balance$. Unlike in [Blakeley *et al.*, 1987], the same relation may appear more than once in a conjunctive query: relations may be joined to themselves. This allows, for example, the grandparent relation to be defined in terms of the parent relation as $\textsf{grandparent} = \textsf{parent} \bowtie_{C=P} \textsf{parent}$.

Given a relational database scheme, a collection of functional dependencies over the relations of the scheme, and a conjunctive query, the decision procedure of this chapter answers the question

> "Does the query necessarily describe an empty set of tuples, for all possible instances of the relations of the scheme that satisfy the functional dependencies?"

The intuitive content of the following theorem is that answering this question is impractical for nonconjunctive relational algebra expressions.

Theorem 4.1: It is an $\mathcal{NP}$-hard problem to check emptiness for a relational expression involving the conjunctive query operators and any single one of the nonconjunctive operators: set union, set difference, or selection based on a disjunctive condition.

Proof: Whether the set of tuples defined by one conjunctive query is always included in the set defined by another conjunctive query is known to be an $\mathcal{NP}$-hard problem [Aho *et al.*, 1979]. It is the case that $A \subseteq B$ if and only if $A - B = \emptyset$. Thus it is $\mathcal{NP}$-hard to decide whether an expression composed using the set difference operator and the conjunctive operators is empty.

Similarly, $A \subseteq B$ if and only if $A \cap B^c = \emptyset$, where B^c is the complement of B. Let the conjunctive query B written in relational algebra notation be $\pi_{FG\ldots}\big(\sigma_{P\wedge Q\wedge\ldots}(R \times S \times \cdots)\big)$ without loss of generality. Then

$$\begin{aligned} B^c = {} & \pi_{FG\ldots}\big(\sigma_{\neg P}(R \times S \times \cdots)\big) \\ & \cup\ \pi_{FG\ldots}\big(\sigma_{\neg Q}(R \times S \times \cdots)\big) \cup \cdots. \end{aligned}$$

If P is an atomic selection condition allowed in a conjunctive query, then so is $\neg P$. Therefore $A \cap B^c$ can be expressed as a query using only the conjunctive operators and the union operator.

Finally, the complement of B can also be written

$$B^c = \pi_{FG\ldots}\big(\sigma_{\neg P \vee \neg Q \vee \ldots}(R \times S \times \cdots)\big)$$

so $A \cap B^c$ can be expressed using only the conjunctive operators and a disjunctive selection condition. ■

Theorem 4.1 is interesting because it pinpoints each of the nonconjunctive relational algebra operators as individually leading to intractability.

4.2 A new tableau notation

The main novelty of the tabular representation for conjunctive relational algebra queries defined here is that the notation can directly accommodate all six comparison operators permitted by Definition 4.1. Clearly the binary operators $<$, $\leq$, $=$, $\geq$, $>$, and $\neq$ are not logically independent. In the same way that formulas of propositional calculus using connectives such as $\wedge$, $\vee$, $\rightarrow$, and $\neg$ can be rewritten to use only connectives from a basis set such as $\{\wedge, \neg\}$, constraints using comparison operators can be rewritten as propositional combinations of constraints using only a subset of the six operators. Previously defined tableaux (for example those of [Klug, 1983]) used $=$ and either $\leq$ or $<$ as their primitive comparison operators. A constraint such as $a \neq b$ was then rewritten as $(a < b) \vee (a > b)$. Unfortunately the introduction of disjunctions cannot be avoided if $\{=, <\}$ or $\{=, \leq\}$ is taken as the basis.

The tableaux defined here use $\{\neq, \leq\}$ as a basis for rewriting comparisons. With this basis, every singleton constraint can be expressed conjunctively, as the following table shows. In rewriting constraints, it makes no difference whether a or b is an attribute variable or a literal constant.

$a < b$	$(a \leq b) \wedge (a \neq b)$
$a \leq b$	$a \leq b$
$a = b$	$(a \leq b) \wedge (b \leq a)$
$a \geq b$	$b \leq a$
$a > b$	$(b \leq a) \wedge (a \neq b)$
$a \neq b$	$a \neq b$

Note that the values of some variables or constants may be drawn from domains for which no natural $\leq$ ordering exists. It is still reasonable to rewrite $p = q$ as $(p \leq q) \land (p \leq q)$, because the decision procedure presented below just uses that pair of comparisons to rewrite one of p and q into the other.

In most previous work, tableaux have been presented using a two-dimensional layout, so the notation of the next definition may appear unconventional. It is used because it facilitates stating transformations on tableaux precisely.

Definition 4.2: A *template* is a list of attribute variables and/or literal constants. A *tableau* is a quadruple $\langle \mathbf{v}, L, N, T \rangle$ where

- $\mathbf{v}$, the *signature*, is a template;
- L is a set of $\leq$ constraints on variables and constants;
- N is a set of $\neq$ constraints on variables and constants; and
- T, the *tableau proper*, is a set of pairs $\langle R, A \rangle$ where R is the name of a relation, and A is a set of templates associated with R. ∎

For example, $\langle B, \mathit{New\ York} \rangle$ is a template, and

$$\langle \langle P, B \rangle, \{\}, \{P \neq \mathit{George\ Bush}\},$$
$$\{\langle \textsf{broker}, \{\langle P, B \rangle\}\rangle, \langle \textsf{office}, \{\langle B, \mathit{New\ York}\rangle\}\rangle\}\rangle$$

is a complete tableau. Tableaux are an alternate notation for relational algebra expressions. As such the formal meaning of a tableau must be the same as that of a relational algebra expression: a mapping from database states (collections of relation instances) to sets of tuples. Before defining the mapping that a tableau denotes, we consider templates. Intuitively a template is a pattern that designates all tuples matching it. This idea is formalized using substitutions: mappings that take variables to attribute values, and constants to themselves. Under a substitution θ the image $\mathbf{a}\theta$ of a template $\mathbf{a}$ is a tuple.

Definition 4.3: The template $\mathbf{a}$ over the relation R defines a mapping $I \mapsto \mathbf{a}(I)$ from database states to sets of tuples such that

$$\mathbf{a}(I) = \{\mathbf{a}\theta : \mathbf{a}\theta \in I(\mathsf{R})\},$$

where θ ranges over all mappings from variables to attribute values, and $I(\mathsf{R})$ is the instance of the relation named R in the database state I. ∎

This definition says that if $I(\mathsf{R})$ is the current extension of R, then the meaning of a template over R is the set of all tuples in $I(\mathsf{R})$ that are substitution instances of the template. Now it is easy to define how a tableau describes a set of tuples given a database state.

Definition 4.4: The tableau $\mathbf{T} = \langle \mathbf{v}, L, N, T \rangle$ defines a mapping $I \mapsto \mathbf{T}(I)$ from database states to sets of tuples such that

$$\begin{aligned} \mathbf{T}(I) = \{\mathbf{v}\theta : & \forall \langle x, y \rangle \in L \;\; x\theta \leq y\theta \\ & \wedge \;\; \forall \langle x, y \rangle \in N \;\; x\theta \neq y\theta \\ & \wedge \;\; \forall \langle \mathsf{R_k}, A \rangle \in T \;\; \mathbf{a} \in A \;\rightarrow\; \mathbf{a}\theta \in I(\mathsf{R_k})\}. \end{aligned}$$

As above, θ ranges over all mappings from variables to attribute values, and $I(\mathsf{R_k})$ is the instance of the relation named $\mathsf{R_k}$ in the database state I. ∎

For a fixed database state, the meaning of a tableau is all substitution instances of the signature of the tableau that satisfy the comparison and membership constraints stated in the tableau.

Every conjunctive query can be translated straightforwardly into a tableau that defines the same set of tuples relative to all database states. First a notation is needed for stating that entries in a tableau should be rewritten to be identical.

Definition 4.5: The notation $\mathbf{T}[A \equiv B]$ is read $\mathbf{T}$ with A and B identified. If $\mathbf{T}$ is a tableau and A and B are variables or constants, then $\mathbf{T}[A \equiv B]$ stands for $\mathbf{T}$ rewritten so each appearance of B is replaced by A, or vice-versa, with the convention that if either A or B is a constant and the other is a variable, then the constant is substituted for the variable.

If $\mathbf{v}' = \langle v_1' v_2' \ldots v_k' \rangle$ and $\mathbf{v}'' = \langle v_1'' v_2'' \ldots v_k'' \rangle$ are templates, then $\mathbf{T}[\mathbf{v}' \equiv \mathbf{v}'']$ stands for $\mathbf{T}[v_1' \equiv v_1''][v_2' \equiv v_2''] \cdots [v_k' \equiv v_k'']$. ∎

The next definition specifies how to translate a relational algebra expression into a tableau. An expression involving a general join of the form $\alpha' \underset{AopB}{\bowtie} \alpha''$ is first rewritten as $\sigma_{AopB}(\alpha' \times \alpha'')$, and a natural join of two relations, $\mathsf{S} \bowtie \mathsf{T}$ say, is rewritten as $\pi_{S \cup T - \{B\}}(\sigma_{A=B}(\mathsf{S} \times \mathsf{T}))$, where A and B are the names used for the common attribute of S and T in their respective schemas S and T.

Definition 4.6: Let α be a conjunctive query in relational algebra notation. Then the corresponding tableau is defined as follows.

- If $\alpha = \langle C_1 C_2 \ldots C_k \rangle$, a tuple of literal constants, then the corresponding tableau is $\langle \langle C_1 C_2 \ldots C_k \rangle, \emptyset, \emptyset, \emptyset \rangle$.

- If $\alpha = \mathsf{R}$, the name of a relation whose attribute schema is $A_1 \cdots A_n$, then the corresponding tableau is $\langle \langle A_1 \cdots A_n \rangle, \emptyset, \emptyset, \{\langle R, \{\langle A_1 \cdots A_n \rangle\}\rangle\}\rangle$.

- If $\langle \mathbf{v}, L, N, T \rangle$ corresponds to α', then $\langle \pi_F(\mathbf{v}), L, N, T \rangle$ corresponds to $\alpha = \pi_F(\alpha')$, where $\pi_F(\mathbf{v})$ is the template $\mathbf{v}$ projected onto the variables corresponding to attributes in the set F.

- If $\langle \mathbf{v}, L, N, T \rangle$ corresponds to α', then $\langle \mathbf{v}, L \cup \{\langle A, B \rangle\}, N, T \rangle$ corresponds to $\alpha = \sigma_{A \leq B}(\alpha')$, where A and B are variables and/or constants.

- If $\langle \mathbf{v}, L, N, T \rangle$ corresponds to α', then $\langle \mathbf{v}, L \cup \{\langle A, B \rangle\}, N \cup \{\langle A, B \rangle\}, T \rangle$ corresponds to $\alpha = \sigma_{A<B}(\alpha)$.

- Similar definitions apply to $\sigma_{A \geq B}(\alpha)$, $\sigma_{A>B}(\alpha)$, and $\sigma_{A \neq B}(\alpha)$.

- If $\langle \mathbf{v}, L, N, T \rangle$ corresponds to α', then $\langle \mathbf{v}, L, N, T \rangle[A \equiv B]$ corresponds to $\alpha = \sigma_{A=B}(\alpha)$.

- If $\langle \mathbf{v}', L', N', \{\langle R_1, A_1' \rangle, \langle R_2, A_2' \rangle, \ldots\}\rangle$ is the tableau corresponding to the expression α', and $\langle \mathbf{v}'', L'', N'', \{\langle R_1, A_1'' \rangle, \langle R_2, A_2'' \rangle, \ldots\}\rangle$ is the tableau corresponding to the expression α'', with the variables of the latter tableau renamed if necessary to be distinct from those of the former, then the tableau corresponding to the expression $\alpha' \times \alpha''$ is

$$\langle \mathbf{v}'\mathbf{v}'', L' \cup L'', N' \cup N'', \{\langle R_1, A_1' \cup A_1'' \rangle, \langle R_2, A_2' \cup A_2'' \rangle, \ldots\}\rangle$$

  where $\mathbf{v}'\mathbf{v}''$ is the concatenation of $\mathbf{v}'$ and $\mathbf{v}''$.

- If $\langle \mathbf{v}', L', N', \{\langle R_1, A_1' \rangle, \langle R_2, A_2' \rangle, \ldots\}\rangle$ is the tableau corresponding to the expression α', and $\langle \mathbf{v}'', L'', N'', \{\langle R_1, A_1'' \rangle, \langle R_2, A_2'' \rangle, \ldots\}\rangle$ is the tableau corresponding to the expression α'', with the variables of the latter tableau renamed if necessary to be distinct from those of the former, then the tableau corresponding to the expression $\alpha' \cap \alpha''$ is

$$\langle \mathbf{v}', L' \cup L'', N' \cup N'', \{\langle R_1, A_1' \cup A_1'' \rangle, \langle R_2, A_2' \cup A_2'' \rangle, \ldots\}\rangle[\mathbf{v}' \equiv \mathbf{v}'']$$

  assuming $\mathbf{v}'$ and $\mathbf{v}''$ have the same length. ■

Lemma 4.2: The tableau corresponding to any conjunctive query has size proportional to the length of the query, and it can be computed in linear time. ■

4.3 Tableau rewrite rules

Once a conjunctive query has been represented as a tableau $\mathbf{T}$, and given a collection of functional dependencies, the task is to decide whether $\mathbf{T}(I) = \emptyset$ whatever the database state I such that the functional dependencies hold. This is achieved by applying the following three conditional rewrite rules. Each of these rules is of the form

$$\text{"if } \phi \text{ then } \mathbf{T} \rhd \mathbf{T}[\alpha \equiv \beta]\text{."}$$

Like an expression in any language, a conjunctive query designates all tuples satisfying a collection of constraints. For example, the conjunctive query $\pi_{PB}(\sigma_{C>New\ York}(\mathsf{broker} \bowtie \mathsf{office}))$ designates all tuples $\langle P, B\rangle$ such that for some C $\langle P, B\rangle \in$ broker and $\langle B, C\rangle \in$ office and $C > New\ York$.

The tableau representing this query makes all the imposed constraints explicit:

$$\langle\langle P, B\rangle, \{New\ York \leq C\}, \{New\ York \neq C\},$$
$$\{\langle\mathsf{broker}, \{\langle P, B\rangle\}\rangle, \langle\mathsf{office}, \{\langle B, C\rangle\}\rangle\}\rangle.$$

Here *New York* is a literal constant and P, B, and C are attribute variables.

Illustration 4.1: Representing a conjunctive query as a tableau.

This notation means that if the condition ϕ is true of $\mathbf{T}$, then $\mathbf{T}$ may be rewritten so that α and β are identified.

The conditional rewrite rules are:

- *The equality detection rule*: If there exist two variables or constants x and y such that both $x \leq y$ and $y \leq x$ are deducible from the $\leq$ constraints of $\mathbf{T}$, then $\mathbf{T} \rhd \mathbf{T}[x \equiv y]$.

- *The functional dependency propagation rule*: Suppose $\mathsf{R_k}$: $F_1 \cdots F_p \rightarrow G_1 \cdots G_q$ is one of the imposed functional dependencies. If there exist templates $\mathbf{a} \in A$ and $\mathbf{b} \in A$ with $\pi_{F_1 \cdots F_p}(\mathbf{a}) = \pi_{F_1 \cdots F_p}(\mathbf{b})$, where $\langle \mathsf{R}_k, A\rangle$ belongs to the tableau proper of $\mathbf{T}$, then $\mathbf{T} \rhd \mathbf{T}[\pi_{G_1 \cdots G_q}(\mathbf{a}) \equiv \pi_{G_1 \cdots G_q}(\mathbf{b})]$.

- *The inconsistency detection rule*: If $c \leq d$ is deducible from the $\leq$ constraints of $\mathbf{T}$, where c and d are two constants such that $c > d$, or $x \neq x$ is deducible from the $\neq$ constraints of $\mathbf{T}$, where x is any variable or constant, then $\mathbf{T} \rhd \perp$.

Note that each of these three rules says that if a condition holds locally somewhere in a tableau, then two templates or variables/constants may be identified across the whole tableau. Essentially, the decision procedure for conjunctive query emptiness is to translate the query into tableau form, and then to apply the rules above. If $\perp$ is produced, then the conjunctive query is always empty. The next chapter discusses how to implement the rules efficiently; here we discuss under what conditions the rules are correct and no other rules are needed.

Definition 4.7: The *chase* of a tableau is the final tableau obtained from it by applying the three rules as many times as possible. ∎

Lemma 4.3: (Well-definedness) The chase of every tableau is defined uniquely and independent of the order of application of the rules.

Proof: A successful application of any rule reduces the number of distinct variables and constants in a tableau, so the rules can only be applied a finite number of times to any initial tableau. The rules commute with each other, so the order in which they are applied has no effect on what tableau is finally produced. ∎

Lemma 4.4: (Soundness) If $\mathbf{T} \rhd \mathbf{T}'$ by any rule, then $\mathbf{T}$ and $\mathbf{T}'$ define the same mapping from valid database states to sets of tuples.

Proof: Straightforward, using Definition 4.5. ∎

The soundness lemma says that if a tableau can be rewritten to $\perp$, then it always describes an empty set of tuples. This converse of this result depends on the assumption that all variables can take values in some dense domain: any domain where for any u and v such that $u \leq v$, there exists a w such that $u \leq w \leq v$.

Lemma 4.5: (Completeness) If all variables range over dense domains and the chase of $\mathbf{T}$ is not $\perp$, then there exists some valid database state I such that $\mathbf{T}(I) \neq \emptyset$.

Proof: If the chase of $\mathbf{T}$ is $\mathbf{T}' \neq \perp$, then it is possible to find a database state I demonstrating that $\mathbf{T}(I) \neq \emptyset$ by assigning values to the variables of $\mathbf{T}'$ that interpolate between the values of its constants, using the denseness assumption. ∎

The prototypical example of a dense domain is the set of rational numbers: between every pair of rationals, say p/q and r/s where p, q, r, s are integers and $q, s > 0$, it is possible to find another rational, for example in this case $(p+r)/(q+s)$. Of more importance for database applications, the set of character strings of finite (but unbounded) length is also dense under any lexicographic ordering. Between every two character strings it is possible to place another string, if the new string is allowed to be longer than the original strings. For example, there is no string of length five that fits alphabetically between the strings *abcde* and *abcdf*, but it is the case that $abcde < abcdef < abcdf$.

The integers do not constitute a dense domain, and neither do strings of a fixed length. Testing conjunctive queries for emptiness with constraints concerning attributes whose domain is the integers is a variant of integer programming, which is $\mathcal{NP}$-hard. The essential difficulty is that a discrete domain allows one to encode disjunctions using what appear to be conjunctions. For example, a choice between two alternate values such as $(x = 2) \vee (x = 3)$ can be expressed as $(x \geq 2) \vee (3 \geq x)$ if the value of x is forced to be integral.

In practice completeness relative to dense domains is satisfactory. Every domain for which comparisons make sense, that is every set with a partial order on its elements, can be embedded in a dense domain that respects the partial order. The rules above are sound for all domains, and only incomplete in a minor way for domains that are not dense.

Consider the conjunctive relational algebra expressions

$$\pi_{PB}\left(\sigma_{C=New\ York}(\mathsf{broker} \bowtie \mathsf{office})\right)$$

and

$$\pi_{PB}\left(\sigma_{C\neq New\ York}(\mathsf{broker} \bowtie \mathsf{office})\right).$$

Suppose the usual functional dependency office : $B \rightarrow C$ is imposed. Then the two queries always describe disjoint sets of tuples. The soundness and completeness lemmas say that the rules above can be used to show this. Consider the tableau representing the intersection of the two queries,

$$\langle\langle P, B\rangle, \{New\ York \leq C, C \leq New\ York\}, \{New\ York \neq C'\},$$
$$\{\langle\mathsf{broker}, \{\langle P, B\rangle\}\rangle, \langle\mathsf{office}, \{\langle B, C\rangle, \langle B, C'\rangle\}\rangle\}\rangle.$$

By the equality detection rule this tableau can be rewritten to

$$\langle\langle P, B\rangle, \{New\ York \leq New\ York\}, \{New\ York \neq C'\},$$
$$\{\langle\mathsf{broker}, \{\langle P, B\rangle\}\rangle, \langle\mathsf{office}, \{\langle B, New\ York\rangle, \langle B, C'\rangle\}\rangle\}\rangle.$$

The two templates $\langle B, New\ York\rangle$ and $\langle B, C'\rangle$ have equal first components, so by the functional dependency propagation rule, *New York* and C' can be identified, giving

$$\langle\langle P, B\rangle, \{New\ York \leq New\ York\}, \{New\ York \neq New\ York\},$$
$$\{\langle\mathsf{broker}, \{\langle P, B\rangle\}\rangle, \langle\mathsf{office}, \{\langle B, New\ York\rangle, \langle B, New\ York\rangle\}\rangle\}\rangle.$$

Finally, the inconsistency detection rule yields $\perp$, since the constraint $New\ York \neq New\ York$ appears above.

Illustration 4.2: Testing disjointness by rewriting.

Chapter 5

The fast chase algorithm

The soundness and completeness lemmas of the previous chapter say that rewriting with its three rules constitutes a decision procedure for the question of whether a tableau defines the empty set of tuples relative to every valid database state. The problem investigated in this chapter is to find a good representation for tableaux and good algorithms for applying the rules.

The first section below summarizes the algorithm and its performance. The following sections describe algorithms for applying each conditional rewrite rule separately, and then explain how to coalesce into one procedure the different algorithms.

5.1 A summary of the algorithm

By the well-definedness lemma, the three conditional rewrite rules of the previous chapter may be applied to a tableau in any order. In principle it is enough to apply the inconsistency detection rule just once, last of all, because applying it cannot create any new opportunities for the functional dependency propagation rule and the equality detection rule to be applied.

The effect of applying either the equality detection rule or the functional dependency propagation rule is to assert equalities between pairs of variables and/or constants. Each rule may be viewed as an operationalized version of a first-order quantifier-free logical theory: the theory of a dense partially ordered set and the theory of a collection of functional dependencies on tuples of uninterpreted symbols (whether these are variables or constants is irrelevant). The two theories do not share any non-logical symbols other than $=$, so any decision procedures for them may be combined using the equality propagation technique of [Nelson and Oppen, 1979]. The insight underlying this technique is that equalities are the only facts deducible by one decision procedure that the other could possibly find useful.

To make applying the equality detection rule efficient, the set of $\leq$ constraints L of a tableau is represented as a directed graph with an edge $a \rightarrow b$ for each $a \leq b$ constraint. Depth-first search partitions this graph in $O(\|L\|)$ time into a directed acyclic graph (DAG) of strongly connected components. The vertices of each strongly connected component are a collection of variables and constants that must be equal. Applying the functional dependency propagation rule is transformed, to the problem of computing the closure of a congruence relation on a graph whose vertices correspond to the components of the templates in the tableau proper T of the tableau. The size of this graph is $O(d\|T\|)$ where d is the maximum number of functional dependencies applicable to any one relation. The parameter d is presumably a small constant. Computing the appropriate congruence closure takes $O(d\|T\| \log d\|T\|)$ time.

The two separate procedures are combined by calling the equality detection procedure each time two variables or constants of L are discovered to be equal by the functional dependency propagation procedure. This may happen at most $\|L\|$ times. Whenever x and y, say, are discovered to be equal, the new pair of edges $x \rightarrow y$ and $y \rightarrow x$ are added to the DAG constructed from L. These edges effectively encode the fact $x = y$ as the conjunction $(x \leq y) \wedge (y \leq x)$. Adding the edge $x \rightarrow y$ to a DAG creates a new strongly connected component if and only if there already exists a path from the vertex x to the vertex y in the DAG. This can be tested in linear time by depth-first search, so all executions of the equality detection procedure require at most $O(\|L\|^2)$ time.

Finally, the inconsistency detection rule must be applied once. The $\leq$ inconsistency condition can be checked in $O(\|L\|)$ time by a variant of the depth-first search algorithm that applies the equality detection rule. Given a set of $\neq$ constraints N, it is only the case that $x \neq x$ is deducible if $x \neq x$ is actually a member of N. The $\neq$ inconsistency condition is thus easy to check in $O(\|N\|)$ time.

The various time bounds and correctness claims of this chapter can be summarized in the following theorems.

Theorem 5.1: (Correctness) The algorithm sketched here correctly computes the chase of any tableau. ∎

Theorem 5.2: (Time complexity) If the number of functional dependencies applicable to each relation is a constant, then the algorithm sketched here uses $O(\|N\| + \|L\|^2 + \|T\| \log \|T\|)$ time on an input tableau $\langle \mathbf{v}, L, N, T \rangle$. ∎

Checking whether there exists a path between two particular vertices in a DAG is known as reachability testing. The $O(\|L\|^2)$ time required by up to $\|L\|$ reachability tests on the graph constructed from L asymptotically dominates the time required by the whole algorithm to test emptiness. It would be useful to improve this aspect of the algorithm, and at first sight it appears reasonable that there should exist a more sophisticated strategy than depth-first search of the whole DAG to test whether a node y is reachable from a node x, perhaps in time proportional to the distance between from x to y or to the diameter of the graph. Alternatively, amortizing the cost of some preprocessing could be expected to allow reachability tests to be done in sublinear time. Preprocessing is certainly useful for trees, which are special cases of DAGs. In linear time, each node of a tree can be labeled with its preorder and postorder numbers. Then a node y

is reachable from a node x if and only if $pre(x) < pre(y)$ and $post(x) > post(y)$, so reachability can be tested in constant time. Surprisingly, testing reachability on DAGs in sublinear time, even amortized over many tests, is a known open problem [Christos Papadimitriou, personal communication].

5.2 Applying the inconsistency detection rule

Two conditions must be checked when applying the inconsistency detection rule to a tableau $\langle \mathbf{v}, L, N, T\rangle$: one concerning L and one concerning N. As explained above, scanning through N sequentially is sufficient to check the latter condition in $O(\|N\|)$ time. This section therefore concentrates on the inconsistency condition concerning L.

Recall that L is a set of $\leq$ facts. These facts entail further $\leq$ facts; L is inconsistent if and only if L entails $c \leq d$ for some pair of constants d and c such that $d < c$. The "only if" claim follows because otherwise, some assignment to the variables of L exists that satisfies the constraints of L. This is true because, by assumption, the values of all constants and variables are drawn from a dense partially ordered domain. (Note that this universal domain may be the disjoint union of a number of dense totally ordered subdomains such as the set of rational numbers and the set of character strings of finite length. If the universal domain is the union of subdomains, to avoid uninteresting erroneous situations, it is assumed that in each $u \leq v$ fact of L, u and v are from the same subdomain.)

Let the symbol $\mid\!\sim$ abbreviate the consequence relation between L and further $\leq$ facts. The question of whether $L \mid\!\sim c \leq d$ for some pair of constants $d < c$ can be answered in $O(\|L\|)$ time by computing upper bounds. Clearly L cannot entail any $\leq$ fact concerning a variable or constant not mentioned in one of the facts stated explicitly in L, so attention can be restricted to just these mentioned variables and constants. Let p be such a variable or constant. Define $ub(p)$ as the least constant d such that $L \mid\!\sim p \leq d$. If no such constant exists, then define

$ub(p) = \infty$, with the convention that ∞ is a constant greater than all others. L is inconsistent if and only if $ub(c) < c$ for some c.

Construct a directed graph D as follows. For every constant or variable a appearing in L, create a vertex and label it with a. For each imposed $a \leq b$ constraint, create an edge from the vertex labelled a to the one labelled b. Now if $L \mid\!\sim p \leq q$ where p and q are different variables and/or constants, then there must exist a path in D from the vertex labelled p to the one labelled q. Depth-first search can partition D in $O(\|L\|)$ time into an induced directed acyclic graph D' of strongly connected components $S_1, S_2, \ldots S_m$ [Aho *et al.*, 1974], where the vertex set of each component corresponds to an equivalence class of variables and/or constants that must be equal. Now upper bounds can be computed readily. The algorithm is the following.

```
for each S ∈ {S1, S2, ... Sm} do
    if S contains a vertex labelled with a constant c
        then for x ∈ V(S) do  ub(x) ← c
        else for x ∈ V(S) do  ub(x) ← ∞
for each S ∈ {S1, S2, ... Sm} in reverse topological sort order do
    suppose ub(v) = a for some v ∈ S
    for each T s.t. T → S ∈ E(D') do
        for x ∈ V(T) do  ub(x) ← min{ub(x), a}
```

The components $S_1, S_2, \ldots S_m$ are in reverse topological sort order if whenever a path exists from S_j to S_k, then $k < j$. According to this ordering S_1 has only incoming edges, if S_2 has an outgoing edge it must be to S_1, and so on. Thus the elements of S_1 are the maximal variables and constants mentioned in L, the elements of S_2 are the next greatest, and so forth. A reverse topological ordering exists because the reduced graph D' of strongly connected components is acyclic, and it can be computed in $O(\|L\|)$ time.

The first loop above initializes the variables and constants of each equivalence class to have the same upper bound. The reader may check that for each strongly connected component S, it does not matter which constant c labelling one of its vertices is chosen. The second loop propagates upper bounds backwards, from each strongly connected component to its immediate predecessors. Once upper bounds have been computed, whether L is inconsistent can be checked by searching D for a vertex with an upper bound less than its label.

5.3 Applying the equality detection rule

Suppose that $L \mathrel{|\!\sim} p \leq q$ and $L \mathrel{|\!\sim} q \leq p$ where p and q are some pair of variables and/or constants that are not identical. Then there must exist paths in both directions between the vertex labelled p and the vertex labelled q in the graph D constructed from L, so $L \mathrel{|\!\sim} p = q$ if and only if the vertices labelled p and q are in the same strongly connected component of D. As noted above, depth-first search can be used to partition D into a directed acyclic graph of strongly connected components with vertex sets corresponding to equivalence classes of variables and/or constants that must be equal, so one multiple application of the equality detection rule requires only $O(\|L\|)$ time.

The functional dependency propagation rule may declare two variables or constants to be equal, say u and v, even though L does not entail this fact. When this occurs, L must be extended to include the constraints $u \leq v$ and $v \leq u$, and the consequences of the extended L must be computed. Given already computed strongly connected components of the old L, when edges in both directions between the vertices labelled u and v are added to $E(D)$, at most one new strongly connected component may be created, by the merging of two or more old components. The depth-first search algorithm to find strongly connected components can readily be extended to update $E(D)$ with these edges and return the equivalence class $S = \{x : (L \cup \{u \leq v, v \leq u\}) \mathrel{|\!\sim} u = x = v\}$ of all variables and constants in L equal to u and v by the arithmetic equality detection rule in time $O(\|L\|)$ also.

5.4 Applying the functional dependency propagation rule

The problem of applying the conditional rewrite rule for functional dependency propagation as many times as possible is transformed here into the problem of computing a congruence closure on a directed graph where each vertex has at most two outgoing edges.

Definition 5.1: Let $\sim_A$ and $\sim_B$ be two equivalence relations over the same set. $\sim_A$ is *coarser* than $\sim_B$ if each equivalence class of $\sim_A$ is the union of some equivalence classes of $\sim_B$. ∎

Definition 5.2: Let G be a directed graph where the successors of each vertex are ordered. Let $\sim$ be an equivalence relation on the vertices of G. The *congruence closure* $\sim^*$ of $\sim$ is the coarsest equivalence relation such that

(i) if $x \sim y$ then $x \sim^* y$; and

(ii) if the corresponding successors of x and y are equivalent under $\sim^*$, then $x \sim^* y$. ∎

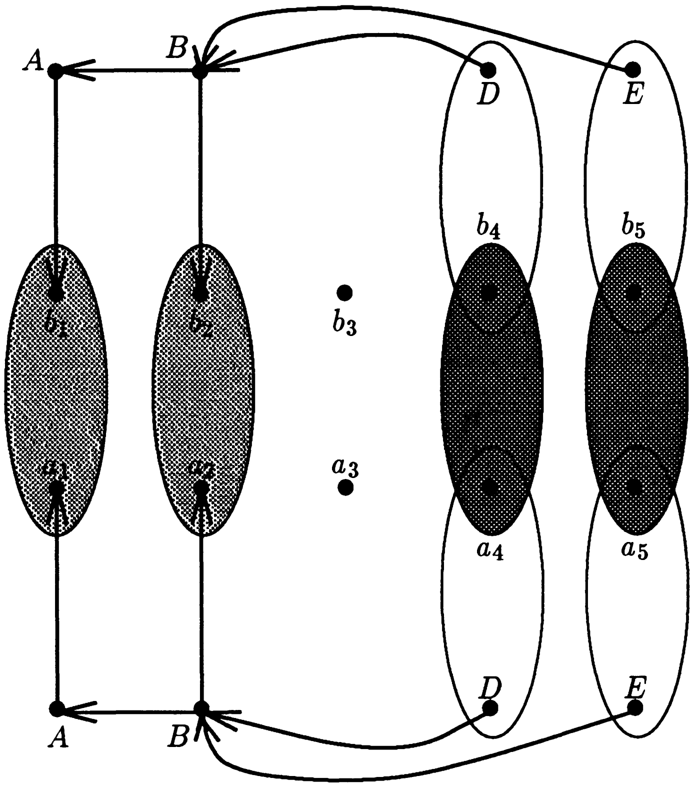
Given a tableau and a collection of functional dependencies that are integrity constraints and hence required to hold in all valid database states, the corresponding directed graph is constructed as follows. The construction here is similar to one in [Downey *et al.*, 1980], but less complicated. Suppose the tableau is $\langle \mathbf{v}, L, N, T\rangle$. Let $\langle \mathsf{R}, A\rangle$ be any element of the tableau proper T, and let $\mathbf{a} = \langle a_1, \ldots a_r\rangle$ be any element of A. The following vertices and edges are created for each $\mathbf{a}$ and each $\langle \mathsf{R}, A\rangle$.

Each component a_i of the template $\mathbf{a}$ is a variable or a constant; it may or may not also appear elsewhere in the tableau. In any case, create distinct vertices with labels a_1 through a_r. Do the following for each functional dependency, say $\mathsf{R} : F_1 \cdots F_p \rightarrow G_1 \cdots G_q$, that is applicable to R, and therefore to the template **a**. Create new vertices with labels F_1 through F_p and G_1 through G_q. Declare each vertex labelled with a schema attribute G_j equivalent to the vertex labelled with its matching template component. Create an edge from each vertex labelled F_i to the one labelled with its matching template component, and an edge from the vertex labelled F_i to the one labelled F_{i-1} for each $i > 1$. Create an edge from the vertex labelled G_j for each j to the vertex labelled F_p. Do this for each template of T, and all functional dependencies applicable to each template. Declare equivalent all the different vertices that are generated from multiple appearances in T of the same constant or variable, and therefore have the same label.

Consider a tableau of the form $\langle \mathbf{v}, L, N, \{\cdots, \langle \mathsf{R}, \{\mathsf{a}, \mathsf{b}\}\rangle, \cdots\}\rangle$ where $\mathsf{a} = \langle a_1, a_2, a_3, a_4, a_5\rangle$ and $\mathsf{b} = \langle b_1, b_2, b_3, b_4, b_5\rangle$ are templates associated with the relation R. Suppose that the schema of R is $ABCDE$ and that the functional dependency $\mathsf{R} : AB \rightarrow DE$ is imposed. Then the templates a and b give rise to the following vertices and edges, with the unshaded ellipses indicating initial equivalence classes.

Illustration 5.1: Propagating dependencies by congruence closure.

Illustration 5.1 (continued)



If the vertices in the lightly shaded ellipses are discovered to be equivalent, then the functional dependency permits the inference that the vertices in the heavily shaded ellipses are equal. Computing congruence closure on the graph performs this inference as follows. The two nodes labelled A have equivalent successors, so they themselves are made equivalent. Then the two nodes labelled B are made equivalent, because they have equivalent first successors, and they already had equivalent second successors. Finally, the pair of nodes labelled D and the pair labelled E are made equivalent for similar reasons. The desired conclusions have been reached.

Lemma 5.3: On a graph constructed as defined above from a tableau **T**, let $\sim^*$ be the congruence closure of of the initial equivalence relation specified. Then $a \sim^* b$ if and only if a and b are identified by repeated application of the functional dependency propagation rule to **T**. ∎

5.5 The combined final chase algorithm

As mentioned before, both the equality detection rule and the functional dependency propagation rule assert equalities between pairs of variables and/or constants. The only predicate used in both theories is $=$, with its standard meaning, so any two decision procedures for them may be combined by communicating $=$ facts in both directions until quiescence. What follows is a description and analysis of the algorithm in Illustration 5.2, which is a simplified version of the congruence closure algorithm in [Downey *et al.*, 1980], extended to account for the extra equivalences that may be asserted by the equality detection procedure.

Given the tableau $\langle \mathbf{v}, L, N, T \rangle$, the first step in the algorithm is to construct two graphs from L and T as described previously. The algorithm then partitions the vertex sets of each graph into a collection of equivalence classes.

A number of auxiliary functions are used to manipulate successor lists and equivalence classes. Remember that each successor list is an ordered list of length at most two. Some of these lists are mapped by a special table to one of the vertices of which they are the successor list. Given a vertex v, $query(v)$ returns the vertex recorded in the table with the same successor list as v, if such a vertex exists, and otherwise returns $\perp$. Similarly, $enter(v)$ finds the successor list of v and records v as its corresponding vertex in the table, and $delete(v)$ removes v if it is recorded in the table.

An equivalence class is represented by associating each member vertex with the name of the class. The name of each class is some canonical vertex that is a member of the class. The name of the class containing a vertex x is obtained by calling the $class(\cdot)$ function with argument x. For each equivalence class, a list is maintained of all the vertices that have at least one successor in the class, in arbitrary order. Given x, $predlist(x)$ returns this list for $class(x)$. Note that operations to update $predlist(\cdot)$ counts, $class(\cdot)$ names, and so on, are not stated explicitly in Illustration 5.2.

Whenever the name of the equivalence class of one successor of a vertex changes, that vertex is added to *pending*, which is a first-in first-out queue. If another vertex has the same successor list, then the equivalence classes of both, together with those of vertices which are equivalent to either by the equality detection procedure are merged. This is where the two procedures are combined: each vertex in $V(G)$ labelled with a constant or a variable also appearing in L is tagged with the vertex naming its corresponding equivalence class in $V(D)$, and vice-versa. Pairs of equivalence classes in $V(G)$ are united following the convention that the name of the one with more predecessors is assigned to the result. Thus whenever two equivalence classes are merged, only the vertices in the shorter predecessor list need be added to the *pending* queue.

The construction of G uses $O(v)$ time, where $v = \|V(G)\|$. There are at most v initial additions of a vertex to *pending*. Whenever a vertex is added to *pending* subsequently, one of the predecessor lists containing it at least doubles in length, and each vertex can be in at most two predecessor lists, so any vertex can only be added to *pending* at most $2\log_2 v + 1$ times. Thus there are only $O(v \log_2 v)$ *query*, *enter*, and *delete* operations on the table of successor lists. For any vertex ever entered into the table, these lists are either singletons or ordered pairs, so the table can be implemented simply by maintaining one array of vertices indexed by $V(G)$, and another indexed by $V(G) \times V(G)$. These arrays use $O(v^2)$ space but there exist techniques that make it unnecessary to initialize them [Aho *et al.*, 1974]. Each *enter*, *delete*, or *query* operation on them requires only constant time. Each equivalence class of $V(G)$ can be tagged to indicate whether it contains an equivalence class of $V(L)$, so L need be updated at most $l = \|L\|$ times. Using the depth-first search procedure described earlier these updates require in total $O(l^2)$ time. Different equivalence classes can be united at most $v-1$ times, but there are potentially $O(v \log_2 v)$ $class(\cdot)$ and $predlist(\cdot)$ lookup operations in all. Using the fast disjoint set union algorithm of [Tarjan, 1975], all the equivalence class operations require only $O(v \log_2 v)$ time in total.

Combining the times found above, the procedure of Illustration 5.2 requires $O(l^2 + v \log_2 v)$ time to compute the chase of a tableau. Suppose at most d functional dependencies apply to the templates over any particular relation schema. Then $\|V(G)\| = O(d\|T\|)$ and $\|E(G)\| = O(d\|T\|)$. If d is a constant, then the time bound is $O(\|L\|^2 + \|T\| \log_2 \|T\|)$.

5.6 Discussion

The first theorem of the previous chapter shows that no efficient chase algorithm exists for expressions drawn from a wider class than that of conjunctive queries, unless $\mathcal{P} = \mathcal{NP}$. However it is possible to extend the algorithm of this chapter slightly. One can deal with additive constants in $\leq$ constraints, while remaining polynomial-time. That is, selection formulas in conjunctive queries may take the form

$$balance \leq salary - 10000.$$

It is also possible to extend the algorithm to deal with generalized functional dependencies as defined in [Maier, 1983]. These extensions are possible because the theory of the rationals under $+$ and $\leq$ and the theory of generalized functional dependencies over uninterpreted symbols each satisfy two conditions: they are decidable in polynomial time, and they are "convex" as defined in [Oppen, 1980].

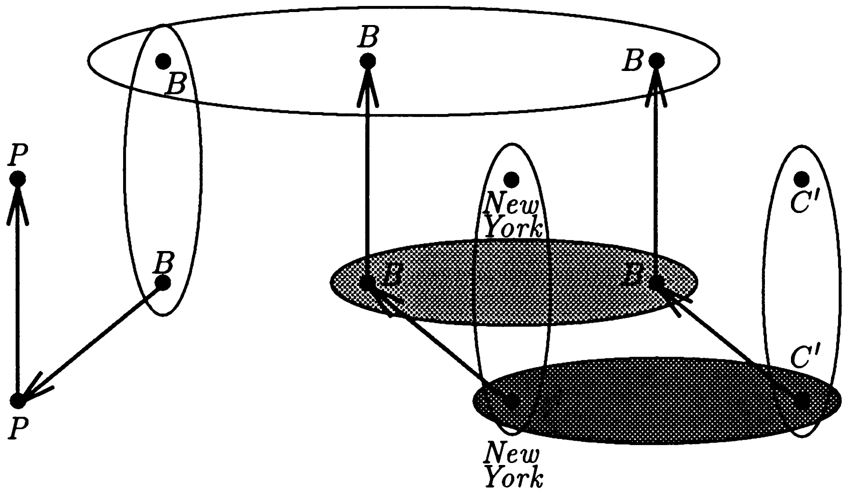
The algorithm described here links and extends a number of strands of related work. Hunt and Rosenkrantz [1979], Larson and Yang [1985], and others have presented algorithms to decide the satisfiability of various classes of selection con-

Input: a tableau $\mathbf{T} = \langle \mathbf{v}, L, N, T \rangle$

Output: equivalence classes such that x, y are equivalent iff they are identified by repeated application of the functional dependency propagation and equality detection rules

```
construct D from L and compute initial equivalence
        classes of V(D) as described in Section 5.2
construct G from T as described in Section 5.3
add each v in V(G) with degree(v) > 0 to pending
while pending ≠ ∅ do
    remove v from pending
    if query(v) = ⊥ then enter(v)
    else
        w ← query(v)
        if v and w in G are tagged with v' and w' in D then
            C ← the new strong component of D with v' and w'
            S ← {z ∈ V(G) : x is tagged with z where x ∈ C}
        else S ← {v, w}
        choose y ∈ S and set S' ← S − {y}
        for each x in S' do
            if ‖predlist(x)‖ > ‖predlist(y)‖ then swap x and y
            for each z ∈ predlist(x) do delete(z)
            append predlist(x) to pending
            merge class(x) and class(y), naming the result class(y)
            concatenate predlist(x) and predlist(y)
```

Illustration 5.2: The extended chase algorithm.



An intermediate tableau of Illustration 4.2 was

$$\langle\langle P, B\rangle, \{New\ York \leq New\ York\}, \{New\ York \neq C'\},$$
$$\{\langle \textsf{broker}, \{\langle P, B\rangle\}\rangle, \langle \textsf{office}, \{\langle B, New\ York\rangle, \langle B, C'\rangle\}\rangle\}\rangle.$$

The directed graph constructed from the tableau proper of **T** is shown above. Labelled vertices contained in the same ellipse are in the same equivalence class. The top three vertices labelled B are equivalent because they are generated from multiple appearances of the same variable. The two vertices inside the lightly shaded ellipse have equivalent successors, so they are themselves equivalent, as the ellipse shows. The vertices in the heavily shaded ellipse therefore have equivalent successors, and are themselves equivalent. No other equivalences are deducible, so the labels of the vertices in the heavily shaded ellipse, namely *New York* and C', are the only components of **T** to be identified by functional dependency propagation.

Illustration 5.3: Applying the chase algorithm.

ditions. The connection established in Section 5.1 with the problem of directed graph reachability is new. Algorithms that compute the consequences of functional dependencies have been published by Downey, Sethi, and Tarjan [1980], and others, but inferring the consequences of selection conditions combined with functional dependencies is new.

Chapter 6

Adaptive locking

As discussed in Chapter 2, every database system that allows transactions to be submitted concurrently by different users must restrict in what order the access commands making up the transactions are executed, if each transaction is always to see the shared database in a consistent state. This chapter discusses how to use the results of the previous chapters to design a concurrency control scheme that meets the flexibility criteria advanced in Chapter 2. The name of the proposed scheme is "adaptive locking." The central idea is that the database system scheduler should automatically generate locking commands on behalf of each transaction. The locks claimed by these commands should cover exactly those portions of the database that it is necessary and sufficient to protect for each transaction's execution to be allowable.

The next section describes the protocol to be followed by an adaptive locking scheduler, and how to check for conflict between transactions using the decision procedure for conjunctive query emptiness. Section 6.2 discusses how to extend adaptive locking for use with complex query languages such as SQL. Section 6.3 compares adaptive locking to conventional schemes, and Section 6.4 discusses its expected performance. Finally, Section 6.5 presents an extended example of a number of concurrent transactions controlled by an adaptive locking scheduler.

6.1 Adaptive locking

The central idea of adaptive locking is simple: the database system scheduler should examine the query and update commands issued by each transaction and preceded each automatically with an appropriate locking command. The locks claimed by these commands should be on exactly those parts of the database that it is necessary and sufficient to protect for the access command to be executable without causing any inconsistency.

Consider an update issued by one transaction, say the command **delete** $\mathsf{R} : \alpha$ where R names a relation and α is a conjunctive query expression. For this command to be executable, it must be independent of all queries already accepted from other transactions. That is, the delete command must not potentially change the answer to any query that belongs to another transaction and has been passed for execution by the scheduler. Let **read** β be such a query, where β is also a conjunctive query expression. Essentially, the two query expressions must always designate disjoint sets of tuples, that is, their intersection must be empty. The next theorem states this criterion precisely.

Theorem 6.1: Let **delete** $\mathsf{R} : \alpha$ or **insert** $\mathsf{R} : \alpha$ be any update, and let **read** β be any query. Suppose that in tableau notation, $\alpha = \langle \mathbf{v}, L, N, T \rangle$ and $\beta = \langle \mathbf{v}', L', N', T' \rangle$. Let the tableau proper of α be

$$T = \{\cdots, \langle \mathsf{R}, \{\mathbf{a}_1, \ldots, \mathbf{a}_k\}\rangle, \cdots\}.$$

The update and the query are independent if for $i = 1, \ldots, k$,

$$\langle \mathbf{v}, L \cup L', N \cup N', \{\langle R_1, A_1 \cup A_1' \rangle, \langle R_2, A_2 \cup A_2' \rangle, \ldots\}\rangle[\mathbf{v} \equiv \mathbf{a}_i]$$

always describes the empty set of tuples.

Proof: By Theorems 3.5 and 3.6, independence holds if conjoining all constraints imposed on tuples of R by the query to the constraints used to select tuples of R for deletion produces an unsatisfiable combination. The two tableaux $\langle \mathbf{v}, L, N, T \rangle$

and $\langle \mathbf{a}_i, L', N', T' \rangle$ are representations of these constraints. By Definition 4.5, the tableau corresponding to the intersection of these is

$$\langle \mathbf{v}, L, N, T \rangle \cap \langle \mathbf{a}_i, L', N', T' \rangle =$$
$$\langle \mathbf{v}, L \cup L', N \cup N', \{\langle R_1, A_1 \cup A'_1 \rangle, \langle R_2, A_2 \cup A'_2 \rangle, \ldots\} \rangle [\mathbf{v} \equiv \mathbf{a}_i].$$

Note that the usual case is that R is not joined to itself in β, and then $k = 1$. For insertion updates, Theorem 3.6 has the extra condition that α should be monotonic, which is satisfied since α is a conjunctive query expression. ∎

The adaptive locking protocol is for the scheduler to place the locking command

$$\textsf{lock exclusive R} : \langle \mathbf{v}, L, N, T \rangle$$

before each update command of the form **delete** R : α or **insert** R : α. Before each query of the form **read** β, the scheduler places the locking command

$$\textsf{lock shared R} : \langle \mathbf{a}_i, L', N', T' \rangle$$

for each R and $\mathbf{a}_i$. All locks are held until the end of the corresponding transaction.

When a lock command is issued, the lock it claims must be tested for conflict with all locks possessed by other active transactions. Theorem 6.1 shows how to use the decision procedure for conjunctive query emptiness of the previous two chapters to implement this testing for conflict. (It is important to remember that two locks concerning different relations cannot conflict.)

Recall that a concurrency control scheme is correct if all transactions are well-formed and two-phased. If an adaptive locking scheduler does not relinquish any lock on behalf of a transaction until the transaction finishes, then all transactions are automatically two-phased. A transaction is *well-formed* if the read sets and the write set of each of its access commands are contained in the collection of parts of the database locked (and not yet unlocked) by the transaction. Under adaptive locking transactions are always well-formed, and this need not be checked, unlike with other locking schemes. Therefore adaptive locking is a correct concurrency control scheme.

A conflict resolution method is a protocol for choosing which transaction to suspend or terminate when a lock claimed on behalf of one transaction cannot be granted because of a lock previously granted to a different transaction. Adaptive locking is compatible with the standard conflict resolution protocols [Gray, 1978]. The most common method is to suspend the transaction claiming the new lock until the transaction possessing the old lock relinquishes it. There are various algorithms for detecting and resolving the deadlocks that arise when this protocol is followed. An alternative protocol is to abort either of the conflicting transactions, undoing all its effects and releasing all its locks, and then to restart it after a randomly chosen time interval. Under this protocol deadlocks can never occur because transactions are never suspended, and because of the random wait, the probability is zero that some transaction is aborted infinitely often.

Under adaptive locking, locks protecting each of its access commands are granted to each transaction *before* executing each command. Flexible conflict resolution protocols are therefore easy to use with adaptive locking. Different transactions can be aborted or blocked, depending on circumstances. For example, one of each pair of conflicting transactions can be queued and its execution resumed later unless the frequency of deadlock is too high, in which case the least advanced conflicting transaction can be chosen as the one to be aborted and restarted after a random wait. Under an item-oriented automatic locking scheme, if locks are granted during the execution of an access command, as items are touched, then it is more difficult to choose dynamically how to resolve conflicts, because partially executed access commands must be rolled back.

It should be clear that an adaptive locking scheduler does acquire sufficiently comprehensive locks, whatever the implementation of access command execution. A database system using adaptive locking may examine more tuples than are strictly necessary to determine the result of an access command, perhaps because appropriate indices are missing or for other implementation-dependent reasons. However, whatever the state of the database, the final effect of the access expression of a command cannot be affected by the existence or non-existence of any tuple not covered by an adaptive locking lock. The final effect can also not be changed by the values of the attributes of any tuple not covered. Thus it is sufficient to acquire locks protecting just those parts of the database that are logically relevant to each access command.

6.2 Adaptive locking for complex queries

It is unrealistic to allow only conjunctive queries in access commands. Most relational database systems provide a basic query language that is at least as expressive as the full relational algebra. In particular, they allow queries to be written using disjunctive operators such as set union. An adaptive locking scheduler can cope with complex access expressions by various heuristic methods. An expression that involves set union or selection based on a disjunctive formula can be rewritten into the union of a number of conjunctive subqueries, in the same way that propositional calculus expressions can be put into disjunctive normal form. Locks can then be issued for each conjunctive subquery separately.

Access expressions involving a set difference operation need special treatment. Consider the command

$$\mathsf{read}\ \pi_B\left(\sigma_{C=New\ York}(\mathsf{office})\right) - \pi_B\left(\sigma_{C=Los\ Angeles}(\mathsf{office})\right).$$

This query might be issued to check that a special case of the functional dependency $B \rightarrow C$ is satisfied. To prevent all view inconsistencies, other transactions must be prevented from deleting or inserting tuples of office with C attribute value *New York* and tuples with C attribute value *Los Angeles*. For locking purposes, a command $\textsf{read}\ \alpha' - \alpha''$ should therefore be treated as $\textsf{read}\ \alpha' \cup \alpha''$.

In addition to the full set of relational algebra operators, practical query languages typically supply aggregating functions such as sum and max. These functions are evaluated by scanning a relation and performing the indicated operation on the specified attribute of its tuples. For example, the total payroll of a department could be found by the query

$$\textsf{read sum}_{salary}\left(\sigma_{dept=CS}(\textsf{employee})\right).$$

The value of an aggregating function depends on the whole current instance of its argument, so for locking purposes an access command such as $\textsf{read sum}_A(\alpha)$ can be treated as $\textsf{read}\ \pi_A(\alpha)$. It may be worthwhile to treat the command $\textsf{read max}_A(\alpha)$ as only requiring a lock that protects the set of tuples described by the expression $\sigma_{A \geq \textsf{max}_A(\alpha)}(\alpha)$.

Adaptive locking can be used with a query language that allows access expressions to be evaluated incrementally. In SQL for example, a so-called "cursor" can be declared to range over the set of tuples defined by a query expression. This expression is not evaluated at the time the cursor is defined. Instead, further SQL commands can advance the cursor and incrementally obtain tuples described by the expression. At the time a cursor is declared, an adaptive locking scheduler can claim shared and perhaps exclusive mode locks on the whole set of tuples that the cursor will range over, since the algorithm that decides whether transactions conflict only manipulates access expressions symbolically. Other strategies may be useful sometimes. If a cursor selects tuples of the current instance of a relation, say R, ordered by increasing values of some attribute, say A, then an adaptive locking scheduler can claim locks protecting the part of the instance of R accessed so far with successive lock expressions of the form $\sigma_{A \leq C_1}(\textsf{R})$, $\sigma_{A \leq C_2}(\textsf{R})$ where $C_1 \leq C_2$, and so on. Because each new lock expression subsumes previous ones, the computational expense of maintaining many locks on R can be avoided without jeopardizing the correctness of adaptive locking: whenever the cursor is advanced, the most recent previous lock can be relinquished immediately after acquiring the current lock.

6.3 Adaptive locking compared to conventional locking

The question of the relative performance of competing schemes for concurrency control is hard to settle. In this section, adaptive locking is compared qualitatively to item-oriented locking schemes that use a hierarchy of lockable items, since a scheme of this class superseded predicate locking in System R [Chamberlin *et al.*, 1981]. Of course, the ultimate test is practical experience. The prototype implementation of adaptive locking discussed in Section 6.4 is not intended to be a practical database system, so it does not provide any answers to questions concerning performance. The standard papers on the performance of different concurrency control schemes (for example [Kung and Papadimitriou, 1979] and [Tay *et al.*, 1985]) are unfortunately not relevant here, because the schemes analyzed in those papers are all item-oriented, unlike adaptive locking.

The most important design issue in item-oriented locking schemes is the size of lockable items. If this *granularity* is fine—for example, if the lockable items are individual tuples—then the cost of checking for conflict between the locks claimed by different transactions is high: essentially, very long bit vectors must be intersected. If lockable items are large, then some transactions that do not in fact have intersecting read or write sets will still request locks on the same items, and so they will be forced to execute sequentially. Under adaptive locking the tradeoff between fine and coarse granularity is circumvented. For each access command, the minimal set of tuples that any scheduler must protect is locked as a whole. Adaptive locking effectively provides dynamically variable granularity. If any item-oriented locking scheduler can allow two transactions to execute concurrently because they do not claim locks on the same item, then an adaptive locking scheduler can also allow the two transactions to proceed concurrently.

Since no single granularity ensures good performance under all conditions [Ries and Stonebraker, 1979], some schemes offer a hierarchy of lockable items of varying size. With such a hierarchy, it is a consequence of what it means for a transaction to possess a lock that claiming a lock on an item implicitly claims a lock on all the items contained in it. However, with item-oriented concurrency control, items are typically designated arbitrarily, for example by the location they are mapped to in a hash table. It is then not feasible to take containment into account while checking for conflict. For this reason, many hierarchical item locking schemes require all items to be locked explicitly, and they provide *warning mode* locks [Korth, 1983]. These can be explained informally by the rule for using them; if the hierarchy of items is a tree, this rule is that a transaction may not request a lock of any mode (shared, exclusive, or warning) on an item unless it holds a warning mode lock on the parent of that item already. Warning modes are not needed under adaptive locking because the algorithm for checking whether locks conflict directly determines whether or not the sets of tuples described by two lock expressions are always disjoint.

Some users of a database system may not require the scheduler to prevent all view inconsistencies. These users may be satisfied with so-called "level 2 consistency," where the result of repeating a read command is not guaranteed to be the same. This is the case if shared mode locks are relinquished immediately after execution of the read command that they protect. The advantage of early unlocking is that more transactions may be able to execute concurrently. It is possible for an adaptive locking scheduler to relinquish shared locks immediately after executing each query command. Adaptive locking can therefore be used by a scheduler that enforces only level 2 consistency for some users.

6.4 An extended example of adaptive locking

To display the reasoning about conflict done by a database system with a scheduler based on adaptive locking, the decision procedure for conjunctive query independence has been implemented in a small prototype database system. The prototype system was built in CPROLOG [Pereira, undated], a widely available and robust implementation of PROLOG. The features of the prototype include

- management of relations as collections of PROLOG facts;
- parsing and evaluation of conjunctive queries written in relational algebra;
- automatic issuing of appropriate locks to transactions.

One of the advantages of adaptive locking is that the scheme makes concurrency control independent of most other aspects of the design of a database system. Therefore many features of a database system for real use by multiple users have not been implemented in the prototype. In particular, there is no interface to concurrent processes at the operating system level. The prototype system runs as a single process accepting a single stream of query and update commands labeled with transaction identifiers.

Illustration 6.1 presents a transcript of a stream of commands and the responses of the prototype system. The scheduler silently precedes each access command with locking commands that protect the read and write sets of the access command. Only when there is a conflict between the commands issued by two transactions does the scheduler make a report. Otherwise, each access command is simply executed. Conflicts between transactions are not resolved by the system. As discussed in Section 6.1, there are many possible conflict resolution protocols to choose between.

Specified in Backus-Naur notation, the syntax of an access command that a user may type is

```
TRANSACTION-ID :
    [ read | [ insert | delete ] RELATION_NAME , ] ACCESS-EXPRESSION
```

Notice that each command is labelled by an identifier. Each different identifier names a separate transaction, and each transaction remains active until an abort or commit command concerning that transaction is processed.

In principle different transactions originate from independent users. The textual sequence of commands in the transcript reflects an arbitrary order on when they reach the scheduler of the database system. Transmission delays could well make this order different from the order in which they were issued by remote users. Of course, the order of access commands inside each transaction must be preserved, since each transaction is allowed to see its own intermediate changes to the database.

Access expressions are written in standard relational algebra notation, as far as this is possible using only ASCII symbols. Specifically, a list of capital letters such as `[P,B,C]` denotes a tuple of attributes. Conjunctive query expressions are formed by composition of primitives on the pattern of

`[dole,schwab]`	a constant tuple
`select(C<>ny, office)`	$\sigma_{C \neq New\ York}$(office)
`project([P,B,C], xxx)`	π_{PBC}(xxx)
`broker#B=B#office`	broker $\bowtie$ office

The particular transactions of Illustration 6.1 refer to the standard three relations used often in examples in the previous chapters:

name	*schema*	*dependency*	*interpretation*
office	BC	$B \to C$	broker B is based in city C
broker	PB	$P \to B$	person P is a client of broker B
owns	PSQ	$PS \to Q$	P owns stock S in quantity Q

What users type as input is in `typewriter script`, all computer output is in sans-serif script, and comments added later are in $\boxed{\textit{boxed italic script.}}$

`?- one: read owns.`

Transaction `one` *reads all of relation* owns.

compiling query ...

[bush,dec,300]

[bush,dupont,400]

[hart,att,300]

[hart,gm,150]

[dukakis,sony,100]

[reagan,att,200]

[reagan,ibm,100]

yes

`?- one: delete owns, select(Q>=300,owns)`

The first three tuples of owns *as printed above are removed.*

compiling query ...

yes

`?- two: read select(P=reagan,owns).`

A tuple such as `[reagan,apple,500]` *might have existed in* owns *before the deletion so this read command causes a conflict.*

lock conflict on owns with one

no

`?- two: read select(Q<300,owns).`

No tuple can be in the set described by `select(Q<300,owns)` *and also in the set described by* `select(Q>=300,owns)`, *so no conflict between the transactions named* `one` *and* `two` *is possible.*

compiling query ...

Illustration 6.1: Adaptive locking transcript.

Illustration 6.1 (continued): Adaptive locking transcript.

[hart,gm,150]

[dukakis,sony,100]

[reagan,att,200]

[reagan,ibm,100]

yes

`?- two: read project([P,B,C], client#B=B#select(C=ny,office)).`

An example of a general conjunctive query.

compiling query ...

[bush,schwab,ny]

[dukakis,rose,ny]

[reagan,schwab,ny]

yes

`?- three: insert client, [dole,schwab].`

Inserting this tuple would make transaction `two` *face a phantom tuple anomaly.*

lock conflict on client with two

no

`?- three: delete client,project([P,B],select(C<>ny,client#B=B#office)).`

This command calls for the deletion from client *of the set of tuples described by* $\pi_{PB}\left(\sigma_{C \neq New\ York}\ (\text{client} \bowtie \text{office})\right)$. *By functional dependency propagation, this set only includes tuples which have not been examined by transaction* `two`.

compiling query ...

yes

`?- three: read client.`

Concurrent read commands never conflict.

compiling query ...

Illustration 6.1 (continued): Adaptive locking transcript.

[bush,schwab]

[hart,merrill]

[dukakis,rose]

[reagan,schwab]

yes

?- `four: read client.`

The changes to the database made by transaction `three` *are not yet committed.*

lock conflict on client with three

no

?- `three: commit.`

The changes made by transaction `three` *are made permanent at the time that it completes execution successfully.*

yes

?- `four: read client.`

Now all other transactions may see the database as updated by transaction `three.`

compiling query ...

[bush,schwab]

[hart,merrill]

[dukakis,rose]

[reagan,schwab]

yes

6.5 The importance of adaptive locking

Adaptive locking appears to be the first value-oriented concurrency control scheme for relational database systems for which efficient basic algorithms exist. It is relatively simple to implement item-oriented locking schemes so that locks are issued automatically. This chapter shows how to do the same with locks that describe protected portions of a database by means of logical expressions. Automatically issuing locks that are guaranteed to protect access commands appropriately circumvents the $\mathcal{NP}$-hard task of checking that each transaction claims appropriate locks on the parts of the shared database that it reads and writes.

An adaptive locking scheduler detects potential conflicts by analyzing the access commands of transactions before the commands are executed. Therefore such a scheduler can remain ignorant of how access commands are executed, and the access routines and data structures of a database system applying adaptive locking can be designed quite separately from the system's scheduler, with only maximal concurrency and efficiency in mind. All read and write commands passed for execution by an adaptive locking scheduler can be executed, without any further checking for conflict, using shared access paths and indexes.

On the whole, designers and users of current relational database systems are satisfied with item-oriented locking. The phantom tuple problem can be solved by placing locks on indexes, and designers are used to coping with interactions between schedulers and data managers. Adaptive locking is therefore perhaps most important as a precursor of concurrency control schemes for deductive database systems. Item-oriented locking is not adequate for database systems with recursive query languages. A successful concurrency control scheme for such a system must necessarily analyze queries and updates logically to determine whether they conflict. The inference rules of Chapter 3 are a starting point for this more general analysis.

Chapter 7

Conclusions

The main contribution of this thesis consists of practical methods for detecting that a command updating a database cannot affect the result of a command querying the database. The previous chapter showed how to apply these methods for reasoning about independence in the design of a concurrency control scheme. This chapter outlines one direction for developing that work, and surveys the importance of what has already been achieved.

Specifically, Section 7.1 informally discusses issues involved in designing a value-oriented concurrency control scheme for a fully distributed database system. Section 7.2 evaluates the results on independence reasoning, and Section 7.3 evaluates the merits of the concurrency control proposals.

7.1 Towards a fully distributed database system.

A database system provides two essential functions: maintaining data over time, and executing query and update commands. A distributed database system is one with the capability to maintain data and execute commands in a coordinated way at a number of dispersed sites. Typically the different sites of a distributed database system communicate by sending messages over a network. The potential advantages of making a database system distributed are clear: increased throughput of transactions per second, and increased reliability in the face of hardware (and ideally, software) failures.

A *fully* distributed database system is one where every system function can be performed at multiple sites. In other words, no aspect of data management or transaction execution is permanently centralized at a particular location. If a database system is not fully distributed, then there exist some failures at some sites that cannot be tolerated. Also, there must be at least one bottleneck in transaction processing that cannot be alleviated by increasing the number of processing sites. To avoid the so-called "wandering bottleneck" phenomenon, where the binding constraint on transaction throughput changes as the system is expanded, a fully distributed database system should have a uniform architecture: every site should be capable of performing all system functions. Then it should be possible to increase the processing capacity of the system incrementally just by duplicating processors and software.

No fully distributed database system has yet been built. Nick Roussopoulos at the University of Maryland has a prototype implementation of a relational database system running on multiple workstations linked by an Ethernet, a type of local area network on which messages are broadcast rather than sent point-to-point. In this system, the software to execute query and update commands is fully distributed. The scheduler, however, is centralized: all access commands, whatever the site at which they are issued, are funnelled to one site that decides which may proceed.

Logical reasoning about queries and updates should make fully distributed concurrency control feasible. Although the result of evaluating a query may be very large, as may be the set of tuples updated by a command, the text of any access command is short. We propose that each site to which a user submits an access command should broadcast the text of that command, and the sites should run a distributed agreement protocol to reach consensus (in the face of failures and time warps) concerning which commands are allowed to be executed.

Permission for a transaction to execute a certain command is itself a piece of information. One suitable fault-tolerant agreement protocol is the scheme proposed by Mei Hsu and Va-On Tam of Harvard University [1990]. This protocol maintains a shared directory of access rights, where an access right is a token that permits its holder to perform some operation on a data item. In the context of computer architecture, access rights have been called capabilities. Hsu and Tam make the unstated assumption that access rights refer to uninterpreted data items, but the correctness of their protocol does not rely on this assumption. It should therefore be possible to use it to manage access rights to parts of a database designated by expressions.

The protocol of Hsu and Tam is not fully distributed. They impose a tree topology on the sites that need to agree as to who holds what access rights. Essentially, the directory of access rights is maintained at the root site of this tree. However the protocol is robust in the face of any site failing, and in particular in the face of a failure at the root site. Intuitively, by telling all other nodes that the root is down, the protocol could become fully distributed.

Using an agreement protocol to manage a distributed "meta-database" of information about access rights gives all transactions and sites common knowledge of what parts of the database each wants to query and update. It should be possible to design sophisticated heuristic conflict resolution schemes under which transactions use their common knowledge to negotiate about restarting and blocking.

7.2 The importance of independence reasoning

If a database system is to support many users retrieving and changing shared data simultaneously, then it is vital to know when a query is independent of an update. Reasoning about independence is also useful if information is distributed geographically, or if relations defined in terms of other relations are stored explicitly. The results of this thesis give methods for proving independence in concrete cases.

The level of abstraction of the definition of independence investigated here is high. This is an advantage, because it makes the results widely applicable. Indeed, the notion of independence is related to the notion of logical irrelevance, which has received attention in artificial intelligence work whose aim is to discover when and how a knowledge base can be reformulated to enable certain queries to be answered more efficiently [Subramanian and Genesereth, 1987]. In particular, workers on irrelevance are interested in discovering during a compilation phase that there are redundant ways of answering a query, so as to eliminate the more expensive ways. The focus of the work reported here is on deciding simple cases of independence quickly during the execution of queries and updates.

The previous work with the most similar aim is [Blakeley *et al.*, 1987], which characterizes when a query specified by a conjunctive expression is independent of a tuple insertion or a deletion defined by a selection formula. The results of Chapter 3 are more general in that they take into account integrity constraints, recursive rules, and arbitrary queries. Taking into account integrity constraints is essential. One reason that concurrency control based on logical reasoning has not been implemented in any major relational database system is that without exploiting integrity constraints, independent queries and updates often cannot be recognized as independent [Chamberlin *et al.*, 1981].

7.3 The importance of adaptive locking

On the whole, designers and users of current relational database systems are satisfied with item-oriented locking. The phantom tuple problem can be solved by placing locks on indexes, and designers are used to coping with interactions between schedulers and data managers. Adaptive locking is therefore perhaps most important as a precursor of concurrency control schemes for database systems with recursive query languages, for which item-oriented locking is not adequate. A successful concurrency control scheme for a deductive database system must analyze complex queries to determine whether they conflict. The inference rules of Chapter 3 are a starting point for this more general analysis.

Adaptive locking appears to be the first value-oriented concurrency control scheme for relational database systems for which efficient basic algorithms exist. It is relatively simple to implement item-oriented locking schemes so that locks are issued automatically. Chapter 6 shows how to do the same with locks that describe protected portions of a database by means of logical expressions. Automatically issuing locks that are guaranteed to protect access commands appropriately circumvents the $\mathcal{NP}$-hard task of checking that each transaction claims appropriate locks on the parts of the shared database that it reads and writes.

An adaptive locking scheduler detects potential conflicts by analyzing the access commands of transactions before the commands are executed. Therefore the access routines and data structures of a database system applying adaptive locking can be designed quite separately from the system's scheduler, with only maximal concurrency and efficiency in mind. All read and write commands passed on by an adaptive locking scheduler can be executed, without any further checking for conflict, using shared access paths and indexes.

Both the basic adaptive locking scheme and its extension to fully distributed database systems rely on an evolution in the relative scarcity of different resources in distributed computer systems. Advances in hardware technology have increased computing capacity (measured in MIPS) faster than network bandwidth (measured in MB/s). The availability of increased computing power can be exploited to perform sophisticated logical inferences about queries and updates, while economizing on network communication capacity by sending the same short messages in broadcast mode to all sites.

Bibliography

[Aho *et al.*, 1974] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[Aho *et al.*, 1979] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalence of relational expressions. *SIAM Journal of Computing*, 8(2):218–246, May 1979.

[Bernstein *et al.*, 1987] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[Blaauw and Duijvestijn, 1985] G. A. Blaauw and A. J. W. Duijvestijn. View locking. In G. Bucci and G. Valle, editors, *Computing 85: A Broad Perspective of Current Developments*, pages 157–165. Elsevier Science Publishers B.V. (North-Holland), 1985.

[Blakeley *et al.*, 1987] José A. Blakeley, Neil Coburn, and Per-Ake Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. Technical Report 235, Indiana University Computer Science Department, November 1987. To appear in *ACM Transactions on Database Systems.*

[Brodsky and Sagiv, 1989] Alexander Brodsky and Yehoshua Sagiv. Inference of monotonicity constraints in Datalog programs. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 190–199, Philadelphia, March 1989.

[Chamberlin *et al.*, 1981] D. D. Chamberlin *et al.* A history and evaluation of System R. *Communications of the ACM*, 24(10):632–646, October 1981.

[Downey *et al.*, 1980] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[Elkan and McAllester, 1988] Charles Elkan and David McAllester. Automated inductive reasoning about logic programs. In Kenneth Bowen and Robert

Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, volume 2, pages 876–892, Seattle, Washington, August 1988. MIT Press.

[Eswaran *et al.*, 1976] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[Gray, 1978] Jim Gray. Notes on database operating systems. Technical Report RJ2188, IBM San Jose, 1978.

[Hunt and Rosenkrantz, 1979] Harry B. Hunt and Daniel J. Rosenkrantz. The complexity of testing predicate locks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 127–133, May 1979.

[Jordan *et al.*, 1981] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision locking. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 143–147, May 1981.

[Klug, 1983] Anthony Klug. Locking expressions for increased database concurrency. *Journal of the ACM*, 30(1):36–54, January 1983.

[Korth, 1983] H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

[Kung and Papadimitriou, 1979] H. T. Kung and C. H. Papadimitriou. An optimality theory of concurrency control for databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 116–126, May 1979.

[Larson and Yang, 1985] Per-Ake Larson and H. Z. Yang. Computing queries from derived relations. In *Proceedings of the International Conference on Very Large Databases*, pages 259–269, 1985.

[Maier, 1983] David Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1983.

[Naughton and Sagiv, 1987] Jeffrey F. Naughton and Yehoshua Sagiv. A decidable class of bounded recursions. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 227–236, San Diego, California, March 1987.

[Nelson and Oppen, 1979] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–252, October 1979.

[NSF, 1990] Database systems: Achievements and opportunities. *Computing Research News*, 2(3):18–19, July 1990. Summary of the Report of the NSF Invitational Workshop on Future Directions in DBMS Research.

[Oppen, 1980] D. C. Oppen. Complexity, convexity, and combination of theories. *Theoretical Computer Science*, 12(3):291–302, November 1980.

[Pereira, undated] Cprolog user's manual, version 1.2, undated.

[Reimer, 1983] M. Reimer. Solving the phantom problem by predicative optimistic concurrency control. In *Proceedings of the International Conference on Very Large Databases*, pages 81–88, September 1983.

[Reiter, 1988] Raymond Reiter. On integrity constraints. In *Proceedings of the Second Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 97–111, Asilomar, California, March 1988. Morgan Kaufmann Publishers, Inc.

[Ries and Stonebraker, 1979] D. R. Ries and M. Stonebraker. Locking granularity revisited. *ACM Transactions on Database Systems*, 4(2):210–227, June 1979.

[Ross, 1989] Kenneth A. Ross. A procedural semantics for well-founded negation in logic programs. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 22–33, Philadelphia, March 1989.

[Sagiv, 1987] Yehoshua Sagiv. Optimizing Datalog programs. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 349–362, San Diego, California, March 1987.

[Shmueli, 1987] Oded Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 237–249, San Diego, California, March 1987.

[Subramanian and Genesereth, 1987] Devika Subramanian and Michael R. Genesereth. The relevance of irrelevance. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 416–422, 1987.

[Tarjan, 1975] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

[Tay *et al.*, 1985] Y. C. Tay, N. Goodman, and R. Suri. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4):415–462, December 1985.

[Ullman, 1982] Jeffrey D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, Maryland, second edition, 1982.

[Ullman, 1988] Jeffrey D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, Rockville, Maryland, 1988.

[Van Gelder and Topor, 1987] Allen Van Gelder and Rodney W. Topor. Safety and correct translation of relational calculus formulas. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 313–327, San Diego, California, March 1987.

[Vardi, 1989] Moshe Y. Vardi. Automata theory for database theoreticians. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 83–92, Philadelphia, March 1989.

[Wheeler, 1986] Traci Ann Wheeler. Unification and predicate locking. M.S. thesis, University of California, Los Angeles, 1986. Published as UCLA Technical Report CSD-880023.